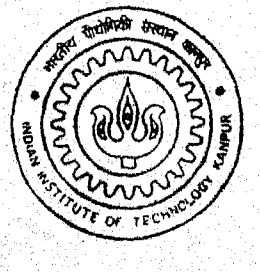
3D Object Realization from Orthographic Projections

by

ANURAG KUMAR JAIN



DEPARTMENT OF MECHANICAL ENGINEERING

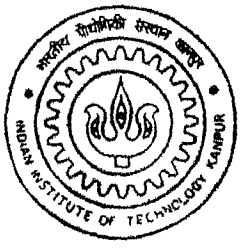
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

February, 2001

3D Object Realization from Orthographic Projections

by

ANURAG KUMAR JAIN



DEPARTMENT OF MECHANICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

February, 2001

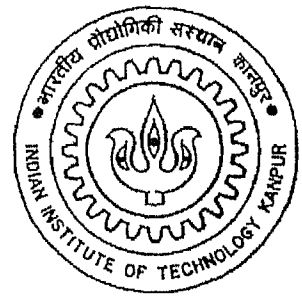
3D Object Realization from Orthographic Projections

A thesis submitted
in partial fulfillment of the requirements
for the degree of

Master of Technology

by

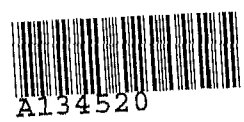
Anurag Kumar Jain



to the

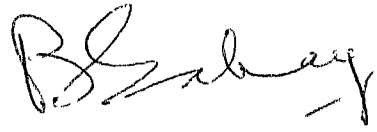
Department of Mechanical Engineering
Indian Institute of Technology
Kanpur-208016, India

February, 2001



CERTIFICATE

It is certified that the work contained in the thesis entitled ***3D Object Realization from Orthographic Projections*** by **Anurag Kumar Jain** *(Roll No:9910503)* has been carried out under my supervision and the work has not been submitted elsewhere for a degree.



Prof. B. Sahay
Department of Mechanical Engineering,
Indian Institute of Technology Kanpur

Feb, 2001

Acknowledgment

The common perception of a thesis is a solitary work, produced by a lone scholar privately toiling away. Nothing could be further from truth. I could not imagine undertaking a task this substantial without help, support and guidance of many people.

First and foremost I would like to express my profound, sincere appreciation and special thanks to my thesis supervisor Dr. B. Sahay for his careful guidance, continuous encouragement and astute suggestions during the entire period of my study at IITK.

I am extremely grateful to Dr S. G. Dhande and the staff of the CAD-P laboratory for providing me all sort of facilities available in their laboratory in support of this thesis work.

I am grateful to all members of the faculty, who imparted knowledge to me by their lectures and gave constant encouragement and support during last one and half years.

I am grateful to Prof. George Markowsky (Chairman Computer Science, University of Maine) for encouraging me and providing his research papers that I could not get at IITK Library.

I would like to thank Mr. Ashwini and other staff members at Drawing Hall Complex for helping me during period of thesis work.

I would also like to express my appreciation to Savy and Ayush for their exellent editorial assistance and patience shown during tedious work of checking the writing work.

Many thanks are due to the friends at IITK for their smiles and friendship making the life at IIT, Kanpur memorable.

Last, but not least, I wish to express my wholehearted gratitude to my beloved parents and sister for their love and blessings. These humble accomplishments are only a small reflection of their tireless and uncompromising efforts. This humble piece of work is dedicated to them.

Name of Student : Anurag Kumar Jain **Roll No. :** 9910503
Degree for which submitted : M. Tech. **Department :** Mechanical Engineeri

Thesis title : *3D Object Realization from Orthographic Projections.*

Name of Thesis Supervisor : *Dr. B. Sahay*

Month and Year of Thesis Submission: February, 2001

Abstract

Developing a solid model with the help of a computer has remained a long cherished dream of mechanical engineers and computer graphics pundits. Representing solids with the help of computers, has tremendous application in many field as it is first step in case of variety of applications like Finite Element analysis. On the other hand most of the engineering drawings are stored and drawn in form of orthographic projections. This is very popular method as it does not require skilled CAD professionals but draftsman who are cheap.

The problem that is dealt here is wire frame reconstruction with the help of orthographic projections The solution presented is robust and computationally very efficient in case of line and arcs but it cannot deal with freeform surfaces. Lots of work has been has been done in this field but most of the algorithm presented till now deals with polyhedral objects. This work can be starting point in direction of development of solid model with the help of drawings.

The algorithm is a frontal tree based search algorithm. Algorithm proceeds in three steps: preprocessing of input data, generating all possible 3D edges from 2D data, redundant edge removal. In second step 2D vertices and their connectivity information is used to generate all possible 3D edges and depends on correspondence that can be established between edges in three views. Edge can be elliptical or circular arc or line. The output after this step will be a set of 3D edges. Redundant edges generated here are removed in later stage , in this step it's more like checking generated wire frame and if some illegal edges are generated they are removed . In fourth step we back project the generated wire frame and find the connectivity of the curved edges to form the surfaces. Curved edges are then faceted to get all edges in form of lines and surfaces are represented as rectangular patches.

Algorithm was tested in case of many complex shapes and results are very promising. Cases dealt with help of this algorithm were much more varied and wide compared to previous works. This algorithm is tested in case of spherical, conical and cylindrical surfaces apart from planar surfaces. The wire frame model developed here can be used further to develop a solid model. This work can be further extended to deal with freeform surfaces.

Contents

List of Figures

Chapter 1

Introduction

In several areas including computer–aided design and manufacturing, automatic representation of solid model with computer is a vital step. Solid representation can directly describe the shape of an object in 3D space which is required as input for numerous applications such as FEM analysis, kinematic simulation, rapid prototyping etc. Two approaches are commonly used for solid modeling

- Primitive modeling
- Translation and rotational sweep

Primitive modeling uses simple 3D objects such as blocks, cylinders, cones etc. A set of regularized *boolean* operations are then performed on these primitives to define complex structures. In *sweep modeling* baseline of an object is used to model a solid object. Geometric model is constructed through either translating the contour of an object along a trajectory, or rotating the cross-section along its center–line.

However these methods are not automatic as they need human interaction. It is also not easy to input data for these methods as most of the input devices are two–dimensional. Apart from that any manufacturing orginization relies heavily on *orthographic projections* to describe mechanical parts. drawings which uses orthographic projections to describe mechanical parts. To solve these problems a more sophisticated approach was devised for constructing solid model. The approach is known as ***solid model reconstruction from orthographic projections*** and it uses 2D data to interpret 3D geometry of the object. The method is used to recover original volumetric information from its planar projections. This is a procedure for obtaining information from lower dimension to higher dimension. Although obtaining 2D projections from a

given 3D object is straight forward but reverse is not easy. The difficulties are due to semantic information loss in 2D drawing. Interpretation of 3D object from engineering drawing involve·

- Human training for interpretation.
- Understanding Conventions and standards.
- Written indirections.

These all features should be incorporated to interpret drawing automatically. In practice, engineering drawing may contain many flaws which human can ignore but machine can not. This makes the problem even more complex. Sufficient methods for automating this step have not been developed despite of its great significance Methods available are computationally expensive. Despite being computationally expensive no method at present is available which can be claimed to be robust. By robust here we mean which can deal with most of the practical cases. These are the reasons why this method is not used by commercial software packages while previous two approaches are used by many despite their drawbacks.

1.1 Literature Survey

Literature available on reconstruction of 3D object from orthographic projection is extensive. Although lots of work has been done in this field since 1970s, but most of the work is concentrated on polyhedral classof objects. None of the available algorithms is able to handle full range of pathological case and ambiguity involved in practical cases. Basically, three dimensional reconstruction algorithm developed till now can be divided into two categories:

- Volume Oriented
- Wireframe Oriented

Volume oriented approach typically uses sweep operations to form primitives. Regularized *boolean* operations are then performed on these primitives to identify final object, but this method is not fully automatic as it requires users help to identify sweep height or sweep angle.

Wireframe oriented approach works bottom–up, starting with a wireframe of 3D edges and vertices reconstructed from 2D vertices and edges. Then real object is found by propagating constraints in this wireframe. This provides a practical way for reconstruction of 3D object from 2D projection and received most attention of researchers.

1.1.1 Volume Oriented Approach

Aldefeld [16-17] was first to propose this algorithm, he viewed complex part as composed of several elementary objects(primitives). These basic primitives were recognized using heuristic search from specific pattern in the 2D representation. However this algorithm can only reconstruct object of uniform thickness. Input for this algorithm needs to specify each elementary objects, this means that each elementary object must be represented as if it were an isolated body.

Bin[15] also proposed a similar algorithm but with some modifications, his programme requires less user interaction and treats a wider range of engineering objects.

Chen[14] proposed an algorithm which consist of three phases: decomposition, reconstruction and composition. This algorithm was also user interactive and can handle polyhedral objects with non–uniform thickness.

Meeran et. al.[11] also proposed a similar algorithm which uses sweep operations to generate basic objects and final object is obtained by combining these with boolean operation. His method deals only with uniform thickness solids.

Tanaka et. al.[3] presented a method of automatically decomposing a 2D assembly drawing into 3D part drawing using a set of solid element equation. Every solid element is classified in one of three types, a true element, a false element and an undecided element. Result produced by this method was able to generate all possible solutions if more than one solution exists. It can deal with only polyhedral class and cylindrical surfaces parallel to one of the axis.

Shum et. al.[1] proposed a two stage extrusion procedure. In each stage, geometric entities from only three orthogonal views (viz. top, side front) are used. In first stage an exterior contour region in each view is swept along its normal direction according to the corresponding object dimension. As a result three extrusion solids are produced. Intersection of these three basic solids form a basic solid. Next, all the interior entities of each view are treated by filtering process. This method has limitation because it uses

linear extrusion and CSG primitive solids such as sphere, cone, torus or tetrahedron can not be generated from linear extrusion.

1.1.2 Wireframe Oriented Approach

Wireframe oriented approach is based on the bottom–up approach proposed by Idesawa[23]. It consist of following steps:

- Generate candidate 3D vertices from 2D nodes.
- Construct 3D edges from 3D vertices.
- Form the face loops of the object from 3D edges.
- Build component of the object from the face loops.
- Combine components to find solution.

He gave a set of rules for identifying edges and vertices in the process of reconstruction. Although his algorithm can deal only with rectilinear objects. Objects with pathological cases or multiple solution cases can not be dealt. It became basis for algorithm devised thereafter.

Wesley and Markowasky[21-22] provide a very thorough analysis of both wireframe generation and wireframe to boundary representation (b–rep) in reconstruction of arbitrary polyhedral solids. It uses detailed view and two or more orthographic view to construct a more precise model. It provides good results in multiple–solution and pathological case but is computationally expensive because it performs a large number of re–projections and complicated geometric operation. It was also limited only to polyhedral objects.

As an extension to Wesley's method, Sakurai[19] proposed an algorithm for reconstruction of cylindrical, conical, torodial and spherical surfaces that are parallel to one of the coordinate axes by using vertex type classification method. Because it is based on Wesley's method it was also complicated and computationally expensive.

Yen *et. al.*[9] presented an efficient method for reconstruction of polyhedral object from orthographic projection based on bottom up approach. Algorithm can handle pathological cases and multiple–solution case. This work formalizes the line constraints idea by defining the qualitative partitioning of search space. To improve the processing

speed of 3D edge generation, the frontal projection based decision tree search technique is used. This work though several year old is one of the best known algorithm for reconstruction of rectilinear solids.

Masuda[4] presented an algorithm based on non–manifold topology and the assumption based truth maintenance system (ATMS). He describes error–recovery system for incorrect orthographic projection using a cellular decomposition model between wireframe and final solid model, which simplifies later stage of conversion to 3D, but the wireframe generation algorithm remains same as Idesawa model.

Shin and Shin[10] presented a solid model reconstruction using geometric properties and the topology of geometric primitives. With this method input of geometric information is easy. However, it takes considerably long processing time as it requires combinatorial search and complicated geometric operations. It can reconstruct object composed of arbitrary planar and limited quadratic faces such as cylinder and tori that are parallel to one of the axes.

Kuo[5] presented a systematic scheme to automatically interpret three–view engineering drawing for quadratic surface solids. Initially a three dimensional wireframe with no face information in it, is reconstructed from orthographic projections. Next all candidate faces are found within the wireframe using minimum–internal–angle search method. Then, pseudo elements are detected using decision chaining method. Finally, all true faces are assembled to form an oriented three dimensional object. Face orientation consistency is ensured using the Moeibus rule. This algorithm is robust but at the same time it is also computationally expensive.

Ah-Soon and Tombre[7] presented first prototype of system combining geometric reconstruction through the "fleshing out projections" paradigm with symbolic recognition and matching. They used algorithm proposed by Markowasky to reconstruct polyhedral object mixed with symbolic information in geometry in different views to "hollowing out" simple mechanical objects such as pins. But this procedure was only able to recognize cylinders(pins) parallel to one of the coordinate axis and was limited to polyhedral object and symbolic reconstruction of pin.

Mukerjee et. al.[2] at **IIT Kanpur** re-considered the problem of converting a linear drawing into a set of 3D edges based on qualitatively complete classification of all possible 3D edges and their projections. The algorithm presented used line–constraint based approach while most of the previous algorithm have considered either point

constraint or a subset of line constraint approach, which showed increase in efficiency. The result presented by them were limited to only polyhedral class of object

Although a lot of work has been done but there is no available software for this purpose presently, which can implement all the stages and can be said to be professional. *Dori and Tombre* [8,12] discussed difficulties in this field and reasons for lack of serious attempt in moving to higher level. And on their experience presented an analysis of mechanical engineering drawing, and a schema is proposed to develop a complete software for achieving high level conversion of technical document of this type. *Harlick and Shaprio* [20] also discussed various aspects of problem and provides the merits and demerits of the key work done in this field and finally proposed research directives to enhance the practicality of paper-engineering-drawing to 3D CAD models. For a complete description of the whole problem reader is directed to read [6,8,12,20].

1.2 Objective of present work

The work presented here aims at efficiently reconstructing 3D wireframe object from its three orthographic projections namely front, side and top view for a more general class of engineering objects. Most of the work done in this field are concentrated on reconstruction of polyhedral objects. Algorithm for reconstructing curved objects at present are computationally expensive and they have a lot of limitations. This work aims at reconstructing wireframe model of objects containing cylindrical, spherical surfaces apart from polyhedral objects.

The algorithm is based on frontal tree based search algorithm proposed by *Yen et. al.*[9] which was for polyhedral class of objects only. Based on this algorithm *Mukerjee et. al.*[2] presented their algorithm which can deal with polyhedral object but was more efficient. First part of our work is to implement the algorithm proposed by them and then develop algorithm based on it for more general class of objects which may have curved surfaces such as object having cylindrical, spherical surfaces. Although this work is not an extension to work done by *Mukerjee et. al.* but basic motivation is work done by them. In the present work we will be proceeding in steps. In first step input data is pre–processed. After preprocessing all the possible 3D edges (arcs and lines) are reconstructed by traversing the front view based decision tree. The edges are reconstructed on the basis of correspondence relationship between edges in

three views. During the next step redundant edges are removed that are either not necessary or duplicated. Finally all curved surfaces are found and some auxiliary edges are added to represent these curved surface. The output is presented in a standard wireframe format So that it can be used as input to solid model generating programme without any further processing, as an extension to this work.

1.3 Organization of Thesis

Thesis is presented in *five chapters.* Chapter two will be a brief review to the various ways of representing engineering objects. Orthographic projections, wireframe representations and B-rep are discussed in brief.

Chapter three discusses the algorithm proposed and various steps involved to reconstruct wireframe objects from orthographic projections.

Chapter four has result and discussion. Here we will be showing the pictorial presentation of the results obtained.

In chapter five we will be discussing about the achievements and limitation of our work. Scope for future work in this field is also discussed in this chapter.

Three appendices are provided. Appendix A describes the input format that is required by the software. Appendix B provides file organization and functions in each file and with their working and arguments. Appendix C is code for edge generation algorithm.

Chapter 2

Engineering Object Representation : A Review

An object has three dimensions namely length, breadth and height. The problem is to represent these objects. There are many representations that are used presently. In this chapter we will have a glimpse over three ways of representation of mechanical parts namely orthographic projection, wireframe representation and boundary representation.

2.1 Orthographic Projections

A projection is a view obtained on a plane by lines drawn to the plane from every point of the object. These lines are called *projectors*. In orthographic projection all the projectors are parallel to each other and perpendicular to the plane of projection. Two planes are necessary in this method of projection, the vertical plane(V.P) and horizontal plane(H.P.) at right angle to each other. These are called principle planes or co–ordianate planes. To obtain the orthographic projections the object is placed in one of the dihedral angle or quadrant made by these two planes. Now, an object can be viewed from six directions front, rear, top, bottom, left and right side so as to obtain six projections on six planes respectively. Among these important views are top view, front view and side view (left or right). An engineering object is commonly represented by these three views. It should be noted that every view gives information about two dimensions: front view gives length and height, top view gives length and breadth, and side view gives breadth and height.

There are two methods of projection, Shown in Figure 2.1 and Figure 2.2. If

object is kept in first dihedral angle, projections obtained are said to be *first angle projection* and if it is placed in third dihedral angle, obtained projections are said to be *third angle projections*. *The Bureau of Indian Standards* has recommended first angle

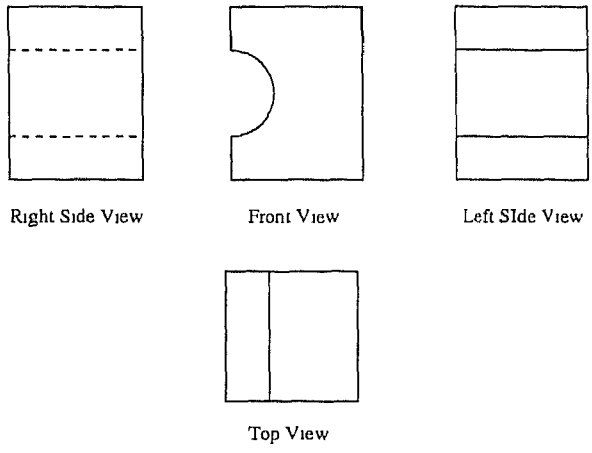


Figure 2.1: First angle projection

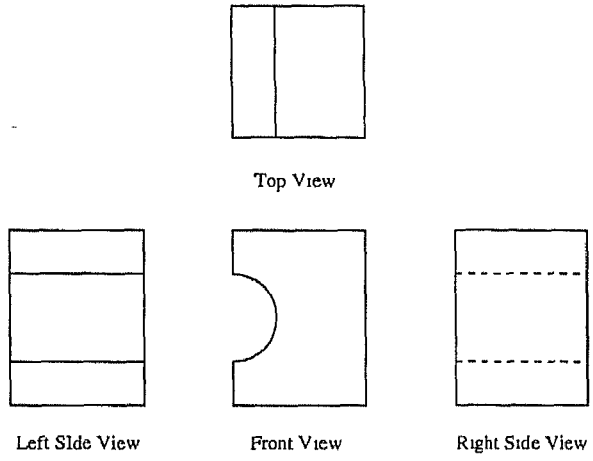


Figure 2.2: Third Angle Projection

projections. However we get a natural view in third angle projection and is used mostly at international level. All the results presented in this thesis are based on *third angle projections*, although results can be obtained for first angle projections also.

2.1.1 Understanding Orthographic projections

For representation of mechanical parts orthographic projections are universally used as they are easy to represent and labour cost associated with it is less. Visualization

requires training, practice and certain basic rules to remember. These rules[27,28] are as follows:

1. A line in a drawing may represent (a) the edge of a surface, (b) intersection of two surfaces (c) the surface limit of the curved boundary.

2. If the line is parallel to the plane of object its projection on the plane will be true length, otherwise it will be less than true length.

3. An area in the drawing may represent a plane or curved surface. If the plane of projection is parallel to object,true shape will be represented, otherwise it will represent shortened surface Two adjacent areas in the drawing represents two different planes.

4. If two lines in space are parallel their projection will be parallel or coincindent in all views, even though they may appear as points in some instances.

2.2 Wireframe representation

A wireframe model of an object is the simplest, but most verbose geometric model that can be represented mathematically on computer. The word wireframe is related to the fact that one may imagine a wire that is bent to follow the object edge to generate the model. Typically a wireframe model consists entirely of points, lines, arcs, circles, conics and curves.

Wireframe model are ambiguous and doesn't provide volumetric information about the object. A wireframe model can provide information about edges, corners, and surface discontinuities but problem associated with this representation scheme is neither it can provide surface information nor it gives spatial addressibility of a point *i. e.* it can not differentiate between inside and outside of an object. The other disadvantage with wireframe model is possibility of creation of nonsense objects[26].

Beside these disadvantages there are lots of advantages also. The major advantage of wireframe model is simplicity of construction. Therefore it doesn't require much processing time and memory as does surface or solid modeling. CPU time required to retrieve, edit or update model is usually small compared to surface or solid model. There are some applications where wireframe input is an ideal input format

such as finite element analysis or numerical control part programming in manufacturing. Printed circuit board design is another area of application of wireframe model. At the same time wireframe model also form basis for surface model. Most existing surface algorithms require wireframe entities to generate surfaces. Some solid modeller such as Medusa and CADD are based on wireframe input. *Markowsaky and Wesley* [11–12] presented an algorithm to discover all the solid from given wireframe model. So if wireframe model can be reconstructed from the orthographic projection it will be useful in many applications and construction of solid model will be the next step. Methods for representing some basic entities such as points, lines and arcs in wireframe are shown in Figures2.3—2.7.

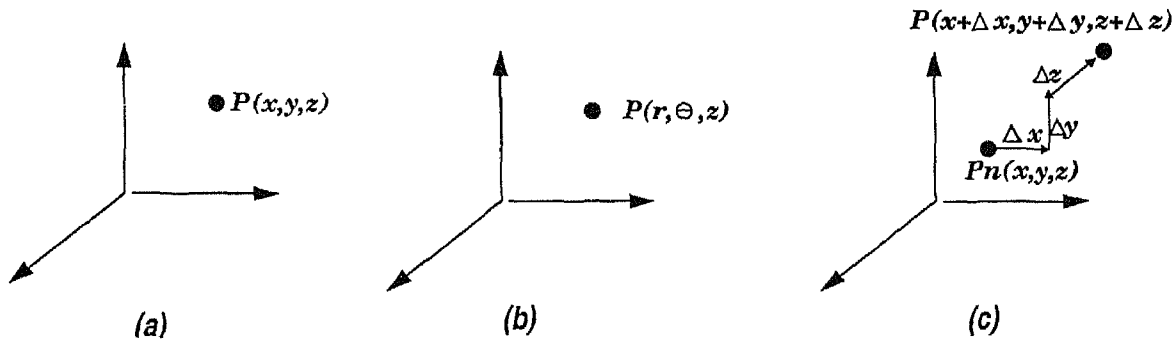


Figure 2.3: Explicit methods of representing points : (a)Absolute Cartesian co-ordinate, (b)Cylindrical co-ordinate, (c)Incremental Cartesian co-ordinate

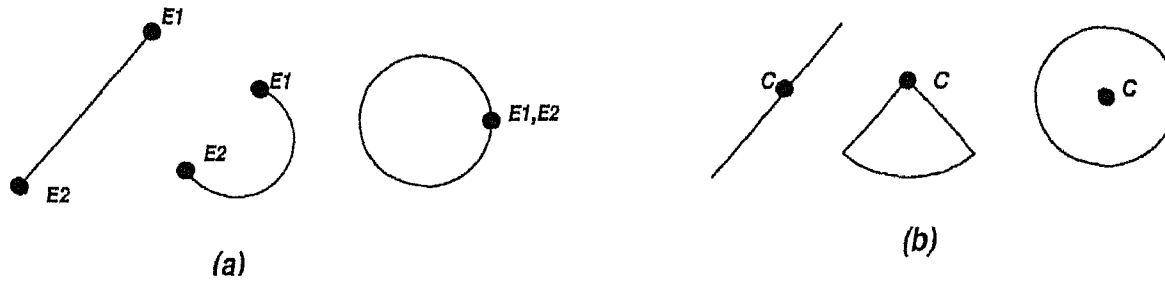


Figure 2.4: Methods of representing points in wireframe : (a) End–point of an existing entity, (b) Center–point of an existing entity

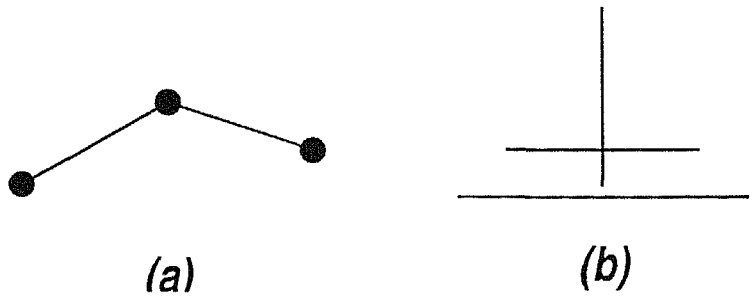


Figure 2.5: Two methods of representing lines in wireframe: (a)By defining points, (b)Perpendicular or parallel to existing entity

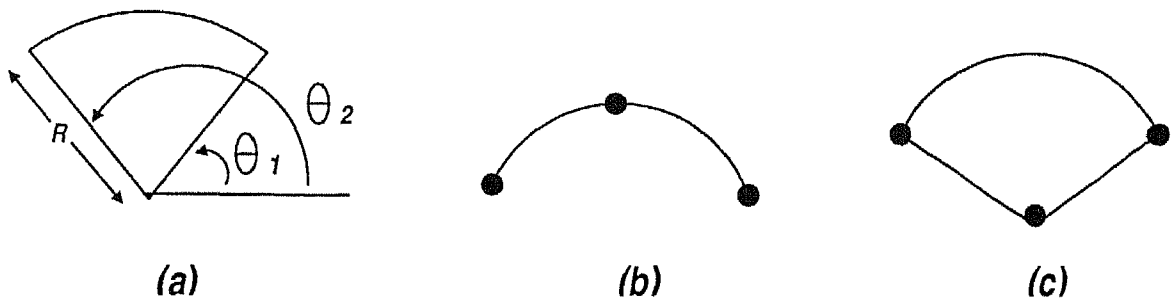


Figure 2.6: Three methods of representing arcs and circles in wireframe: (a)By center point, radius and two angles (b)Three points method (c) Center point and two end points

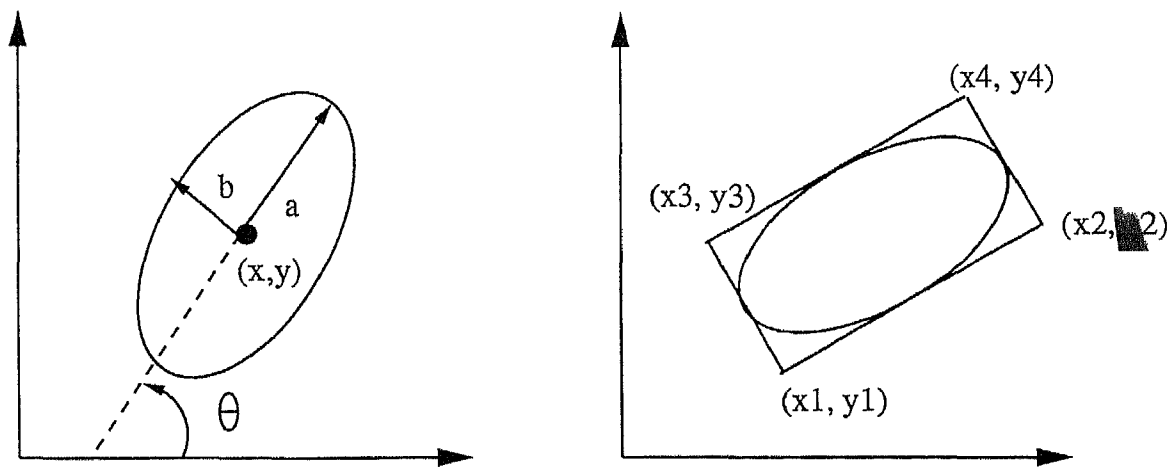


Figure 2.7: Two methods of representing ellipse in wireframe

2.3 Boundary Representation (B-rep)

A B-rep model is based on the topological notion that a physical object is bounded by a set of faces These faces are region or subset of closed and orientable surfaces. A closed surface is one that is continuous without brakes. An orientable surface is one in which it is possible to distinguish between two faces. Each face is bounded by an edge, an edge is bounded by vertices. Thus topologically a boundary model consists of faces, edges and vertices linked together in such way so as to ensure topological consistency of model.

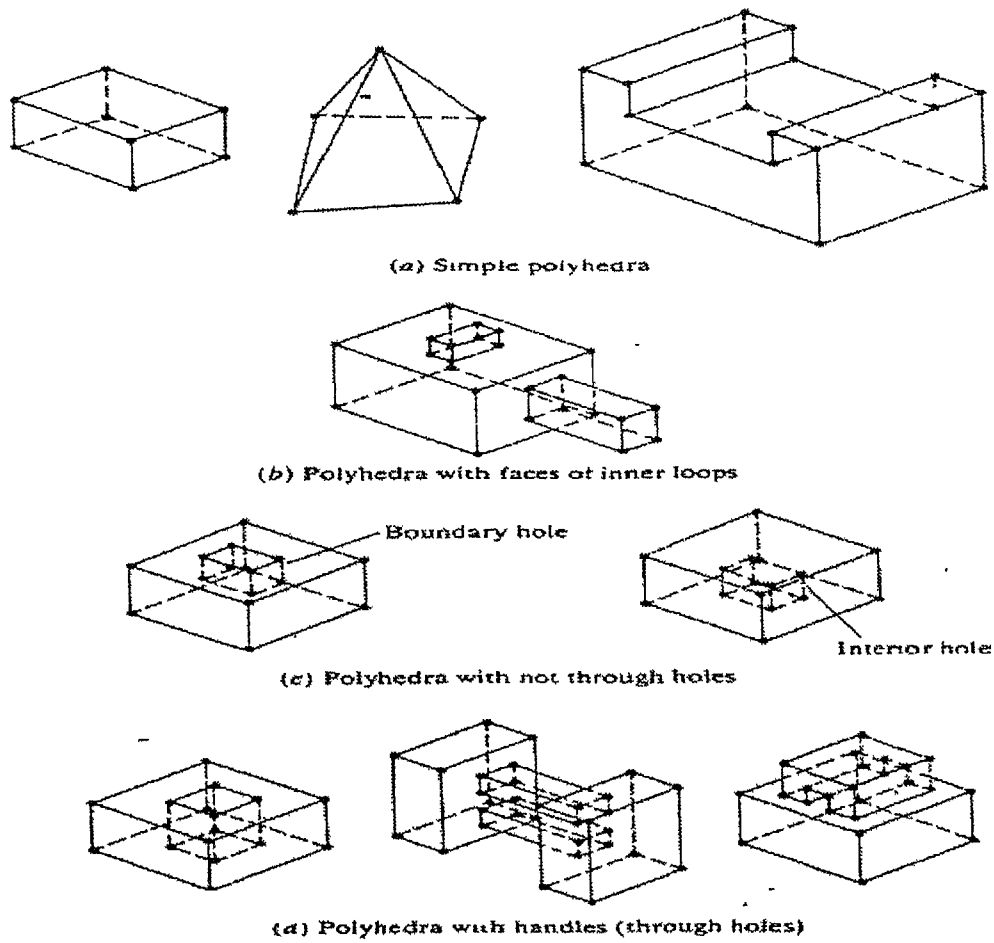


Figure 2.8: Types of Polyhedral Objects as classified by Zeid[25]

Objects that are encountered in engineering can be classified into two categories polyhedral and curved. Polyhedral objects consists of planar faces connected at edges which are in turn connected at vertices. A curved object is one that consists of curved edges as well as linear edges. The representation of curved edges is more complex.

They can be represented in direct and indirect schema. In direct schema edge is represented by curve equation and end points. The edge is represented by intersection of two surfaces In practice indirect schema is preferred because intersection of two underlying surfaces of two faces produces the curved edges of two faces.

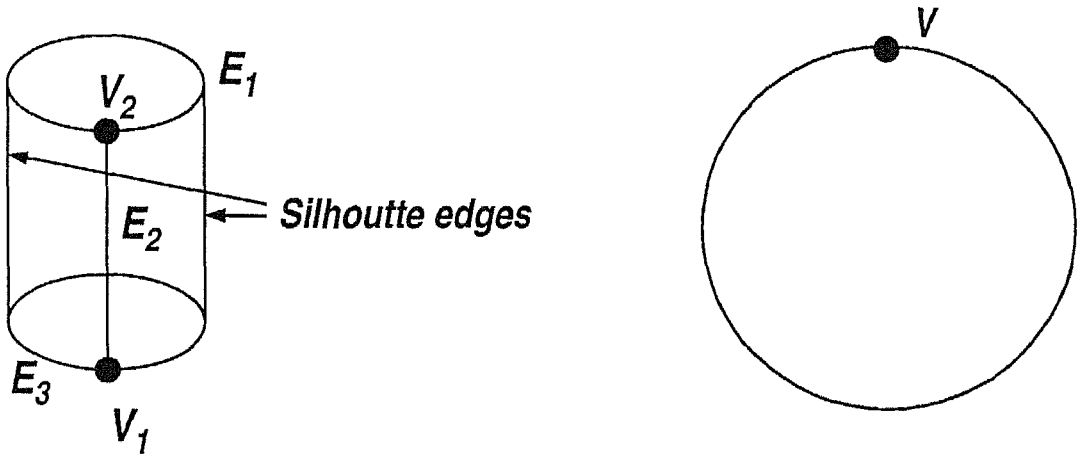


Figure 2.9: Exact representation of curved surfaces

If curved objects are presented by storing the equations of underlying curves and surfaces then boundary schema is known as exact B-rep schema. Exact B-rep for cylindrical and spherical surface are shown in Figure2.9 Another alternative is the approximate or faceted B-rep schema. Figure2.10 shows a faceted B-rep of a cylinder and a sphere. The faceted cylinder is generated by rotating a line incrementally about the cylinder axis, to obtain desired number of facets. The faceted sphere is formed in a similar way by rotating m connected line segments about the sphere axis for a total of n sides. A general data structure for a boundary model should have both topological and geometric information. A winged edge data structure is most useful. In this structure, all the adjacency relations of each vertex and that of each edge are described explicitly. Since an edge is adjacent to exactly two faces it is a component in two loops for each faces. We shall be using winged edge data structure for representation of wireframe. The cylindrical surfaces generated will be presented both in exact and faceted B-rep model.

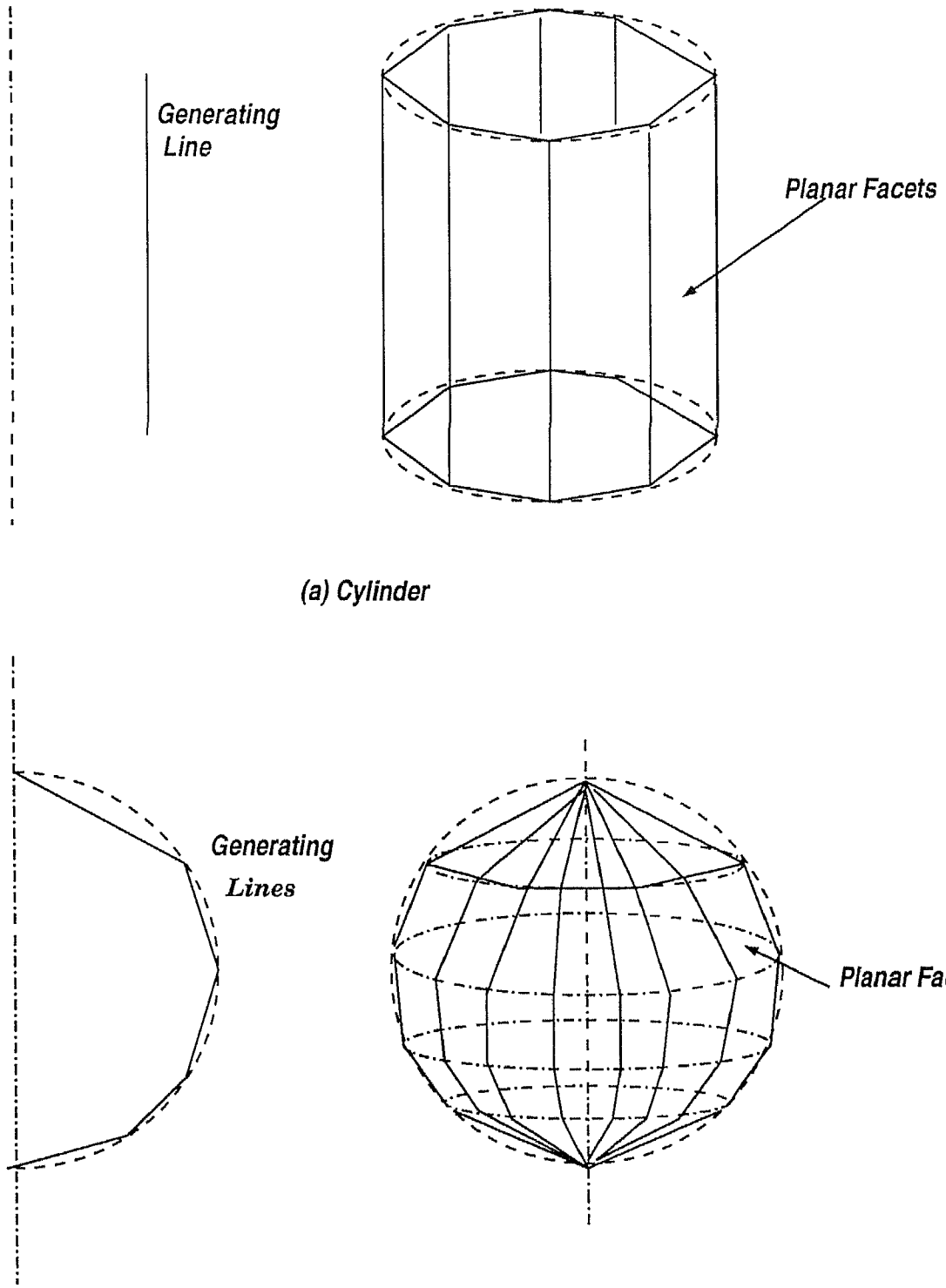


Figure 2.10: Faceted representation of curved surfaces[25]

Chapter 3

Algorithm

In this chapter, the proposed algorithm for reconstruction of 3D model from 2D drawings is discussed. We can represent orthographic projections of a 3D object as mapping from 3D space to 2D. Let f be the mapping function from object O to the projections Ps, this can be represented as:

$$Ps=f(O)$$

Thus 3D model reconstruction from its orthographic projections is to find a inverse function f^{-1} such that :

$$O=f^{-1}(O)$$

This inverse function incorporates two functions, first is to generate wireframe model from 2D data and second is to construct solid model from this wireframe information. As mentioned earlier, only wireframe reconstruction algorithm is discussed here. The algorithm can be divided into four parts :

Step I: *Pre-processing of input data.*

In the first step input data is processed to generate all 2D vertices which are not explicitly mentioned in input data. These vertices are then sorted so as to increase efficiency in succeeding steps.

Step II: *Generating all possible 3D edges from 2D data.*

In the second step, which is the most important step of the procedure, information from 2D vertices and their connectivity is used to generate all possible 3D edges. This relies on a certain correspondence that can be established between three views because each view is a 90^o spatial rotation of the object they represent. A

3D edge is generated, if three edges in each view are such that X co–ordinate of the edges in the front and top views are equal, similar condition for Y co–ordinate in side and top views and Z co–ordinate in front and side views. By an edge here we mean elliptical or circular arc, or line. The output after this step will be a set of 3D lines and arcs which may be again circular or elliptical.

Step III: *Redundant edge removal.*

Second step may generate some illegal edges or redundant edges. Third step is concerned with removal of all the overlapping edges, pathological edges, dangling edges etc. So its like a check on generated wireframe and if some illegal edges are generated they are removed in this step.

Step IV: *Cylindrical, conical, surface representation.*

Nature of surface connecting the edges cannot be drawn from this information. So some auxiliary edges are to be generated so as to represent surface in this wireframe. All the curved surfaces are then searched by back projecting the arcs to orthographic planes. Here we obtain exact B-rep for surfaces. These surfaces are then faceted so as to get faceted B-rep for these surfaces. The final output is vertices in 3D and their connectivity, where type of connectivity is linear. This kind of output is helpful in constructing solid model from this wireframe model. The output is presented in two formats so that it can given direct input to the programme generating solid model with no pre–processing required to be done on this output.

3.1 Pre–processing

This step consists of two parts :

- Generating all the vertices which are not mentioned explicitly in the input.
- Sorting of vertices in each view.

Input may not consist of information of all the vertices in a particular view. Some vertices may be implicit such as intersection of two or more edges. This may result in loss of some edge information in 3D space. Intersection point of all the edges in each view is found out. The possible edges are then generated from these new vertices with

the constraint that these generated edges should be subset of the intersecting edges. Let $E_1\{V_1, V_2\}$ and $E_2\{V_3, V_4\}$ be two edges and V_5 be their intersection point then generated possible edges are

$E_{g1}\{V_1, V_5\}, E_{g2}\{V_5, V_2\} \subseteq E_1$

$E_{g3}\{V_3, V_5\}, E_{g4}\{V_5, V_4\} \subseteq E_2$

The generated edges and vertices are then added to the vertex and edge list respectively of their corresponding view

The vertices are then sorted lexicographically as ordered pairs of end–points, edges in top–view in increasing order of X value and those in side view in increasing order Z value. The efficiency of further computation is improved on the basis of this sorting. The edges are classified into 6 cases and are labeled in this step, these cases will be explained later.

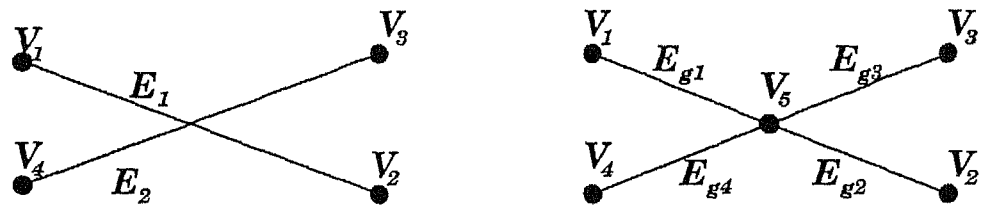


Figure 3.1: An Example of generation of implicit edges

3.2 3D Edge Generation

This step is the heart of the algorithm. Here, all the possible 3D edges on the basis of correspondence between the 2D edges in the three views are generated. 3D edges are generated by mapping from the 2D edges. Both Line and arc segments are considered as an edge in this section.

Let *vertex_list*($\star$) be a 2D vertex list, where ($\star$) can be substituted by F, T, S to represent vertices from front, top and side view respectively. Let *edge_list*($\star$) be a 2D edge list of corresponding projections. Here information regarding edges in each view is stored. Information recorded in a member of this list are:

- Two end points of edge.
- Type whether it is of type "line" or "arc".

- Arc information if it is of type arc.
- Case in which it is classified.

First three are straight forward but third information is recorded during pre-processing step, term *Case* will be explained later.

The algorithm used here is a frontal tree based search algorithm and was first proposed by *Yen et. al.*[9] for line reconstruction only. In contrast to that, our algorithm can deal with arcs and is computationally efficient as compared to that of *Yen's*. In frontal tree based search we start from edges in *edge_list(F)* and then based on their correspondence with edges in *edge_list(S)* and *edge_list(T)* possible 3D edges are generated. Search algorithm is based on classification of the edges in three views. Projected 2D edges are classified as :

- *Case 1:* Line parallel to x axis.
- *Case 2:* Line parallel to z axis.
- *Case 3:* Line parallel to none of the principle axes.
- *Case 4:* Line parallel to two axis i.e it is point.
- *Case 5:* Line parallel to y axis.
- *Case 6:* Arc.

Now 3D edges can then be classified into eight states in which they can exist in space. These states are :

- *State 1:* Line parallel to x axis.
- *State 2:* Line parallel to y axis.
- *State 3:* Line parallel to z axis.
- *State 4:* Line Parallel to none of the axes.
- *State 5:* Arc in a plane parallel to XY plane.
- *State 6:* Arc in a plane parallel to XZ plane.

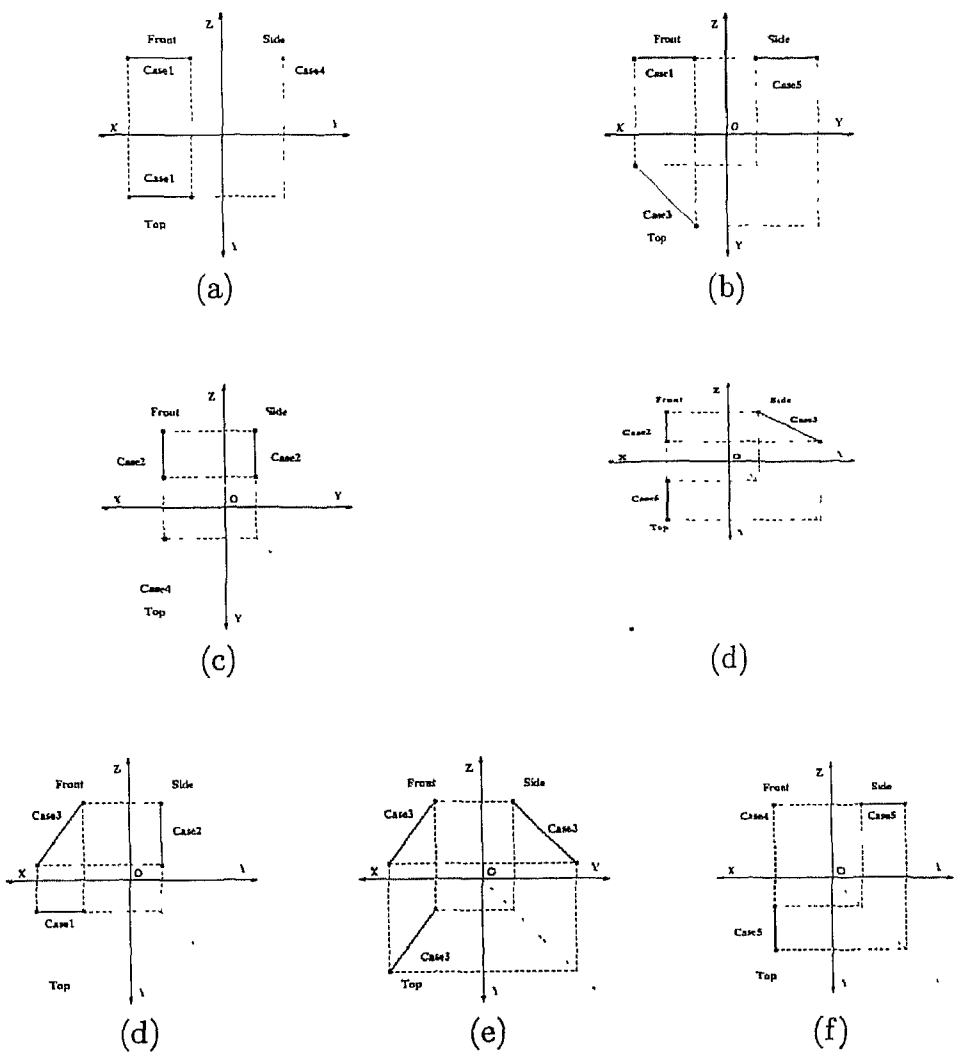


Figure 3.2: Seven possible combinations in which a line in three dimensional space can be projected onto three orthographic planes as presented by *Amitabha Mukarjee et. al.*[2]

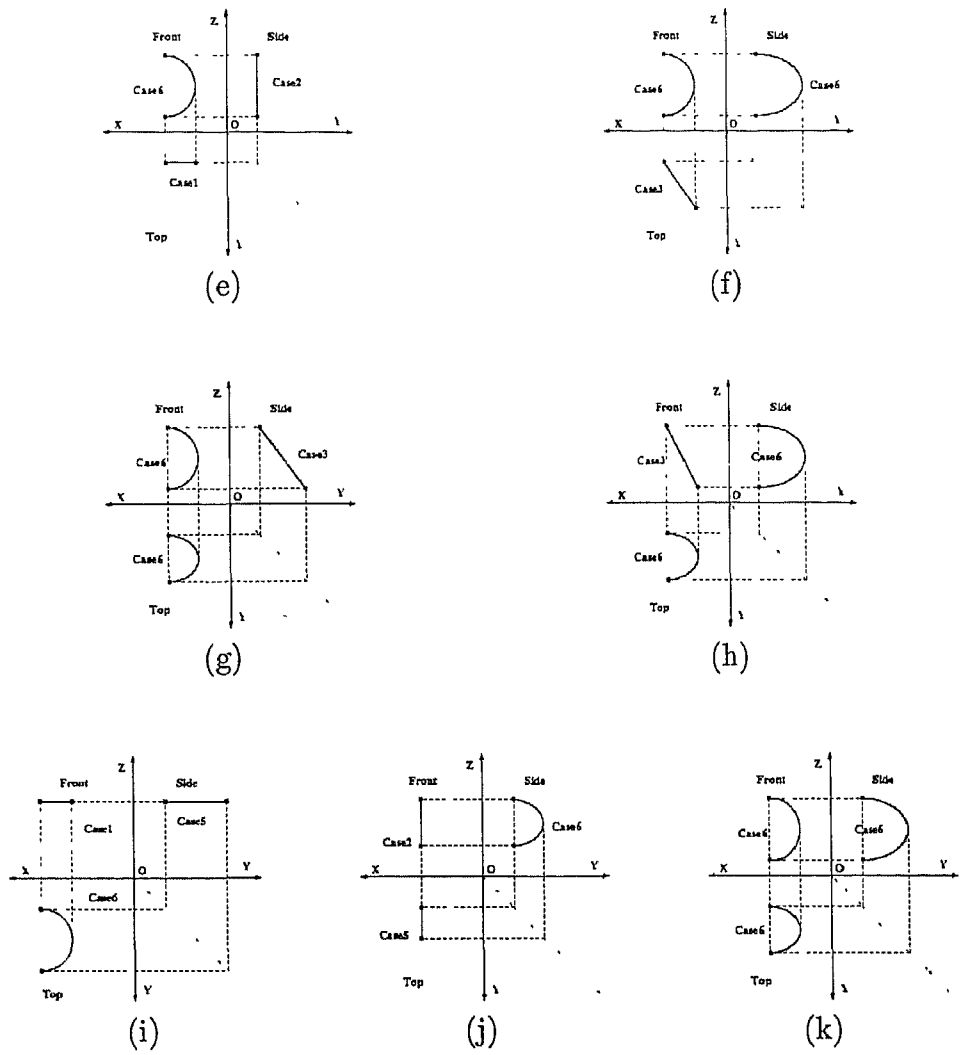


Figure 3.3: Seven combinations in which an arc in three dimensional space can be projected onto three orthographic planes

- *State 7:* Arc in a plane parallel to YZ plane.

- *State 8:* Arc in a plane parallel to none of principle planes.

Based on the above classification schema of edges in 2D projection, it's possible to have only 14 combinations such that edge in one view has *correspondence relationship* to edges in other views and thus form a valid 3D edge There are seven combinations for four states of line in space and similar for arcs. `There is absolutely no other way in which a 3D edge can be mapped into 2D orthographic projection.` Combinations for generating a valid 3D line and that to generate valid 3D arc are as shown in Figure3.2 and Figure3.3 respectively.

3.2.1 Correspondence Relationship

If f_e, s_e, t_e are the projection of same 3D edge in three different views then there must be some relationship between them, this relationship is denoted here as **correspondence relationship.** Let us take a front edge f_e and a side edge s_e. If minimum and maximum value of Z co-ordinate in front edge are equal to the corresponding values in side edge, then there exists a 3D edge such that its projection on XZ plane and YZ plane will result in f_e and s_e respectively. So the correspondence relationship between an edge in side view to that of an edge in front view is :

$$\begin{bmatrix} min(p_f.z) \\ max(p_f.z) \end{bmatrix} = \begin{bmatrix} min(p_s.z) \\ max(p_s.z) \end{bmatrix} \qquad \begin{cases} p_s \in e_s \\ p_f \in e_f \end{cases}$$

i. e. maximum and minimum value of Z co-ordinate must be equal between two edges. Similarly *correspondence relationship* for the edges of front and top view is.

$$\begin{bmatrix} min(p_f.x) \\ max(p_f.x) \end{bmatrix} = \begin{bmatrix} min(p_t.x) \\ max(p_t.x) \end{bmatrix} \qquad \begin{cases} p_f \in e_f \\ p_t \in e_t \end{cases}$$

and for edges of side view and top view combination

$$\begin{bmatrix} min(p_t.y) \\ max(p_t.y) \end{bmatrix} = \begin{bmatrix} min(p_s.y) \\ max(p_s.y) \end{bmatrix} \qquad \begin{cases} p_s \in e_s \\ p_t \in e_t \end{cases}$$

Minimum and maximum co-ordinate is straight forward for line segment where one of the vertex is minimum and other is maximum. But for arc segment it depends on arc angle and its position. Minimum z co-ordinate for front view is shown in Figure 3.4

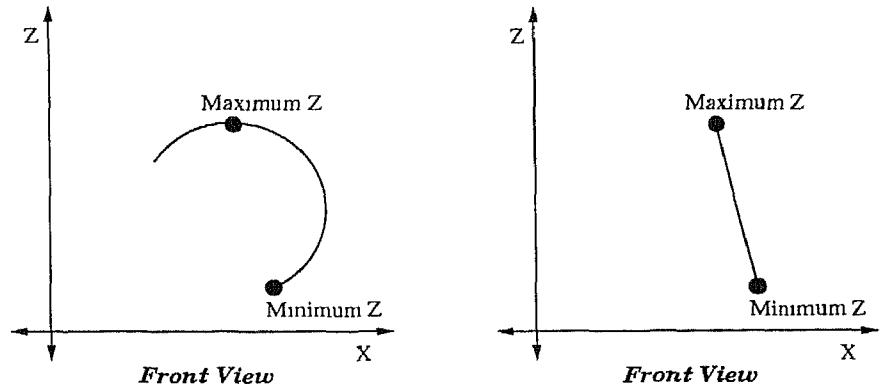


Figure 3.4: Maximum and minimum coordinate value for an edge.

3.2.2 3D edge generation algorithm

A front view based decision tree is then traversed according to 14 combinations of edges discussed previously and *correspondence relationship* between edges in three views. The 3D edges can be generated by traversing this decision tree using algorithm discussed below. The symbols used in this algorithm are as follows:

- f_e= Selected edge from *edge_list(F)*
- t_e= Selected edge from *edge_list(T)*
- s_e= Selected edge from *edge_list(S)*
- f_v= Selected vertex from *vertex_list(F)*
- t_v= Selected vertex from *vertex_list(T)*
- s_v= Selected vertex from *vertex_list(S)*

Step 1: Select an edge f_e from *edge_list(F)*. It can have label either case 1, case 2, case 3, case 4, or case 6 because it is in XZ-plane. If all the edges in *edge_list(F)* are exhausted then go to step 7.

Step 2: Select an edge from *edge_list(T)* if f_e is labeled case 1, case 3 or case 6 and if it is labeled as case 2 select edge from *edge_list(S)*. Select only from the edges that are labeled with case no. corresponding to their position in decision tree shown in Figure 3.5. If there are no more edges in selected edge list go to step 1.

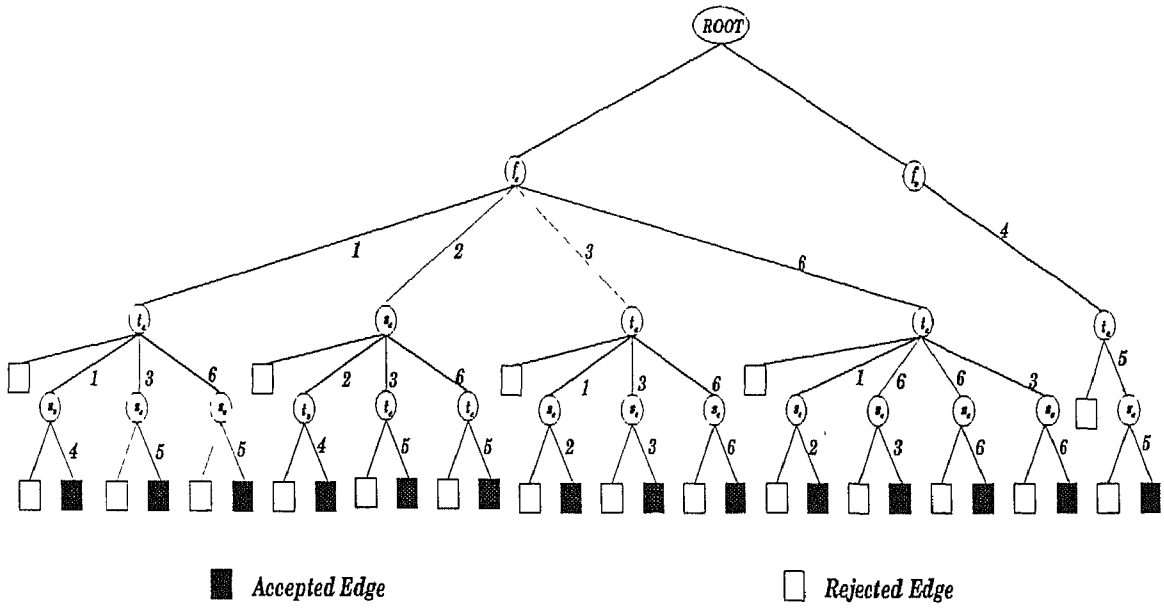


Figure 3.5: Front view based decision tree

Step 3: Check for correspondence relationship between this edge with f_e. If it exists mark this edge t_e or s_e depending on whether it belongs to *edge_list(S)* or *edge_list(T)* respectively. Else go to step 2.

step 4: Select edge from *edge_list(T)* or *edge_list(S)* or a vertex *vertex_list(T)* or *vertex_list (S)* depending on the third level node of decision tree labeled with number corresponding to decision tree. If there is no more item in the selected list then go to step 1.

step 5: Check for correspondence relationship between edge selected in step 1, step 2 and step 4. If correspondence relationship exists mark it s_e or t_e depending upon selected edge list in previous step else go to step 2.

step 6: Make a 3D candidate edge e_c edge with selected f_e, s_e and t_e. Add this to 3D candidate edge list *edge_list(3D)* and go to step 1.

step7: (a) Select a vertex f_v from front vertex list *vertex_list(F)*. If all the vertices in this list are exhausted–**STOP**

(b) Select edge from *edge_list(T)* labeled as case 5. Check correspondence relationship between f_v and this edge, if it exists then mark this edge as t_e, else go to (a).

(c) Select edge from *edge_list(T)* with label case 5. If there is no more edge in this list goto (b). Check correspondence relation between this edge and *Vf*. If it exists mark this edge as e_t and make 3D edge from f_v, s_e, t_e add this to *edge_list(3D)* and go to (b).

3.3 Redundant edge removal

The output after edge generation algorithm discussed above, will be a set of probable edges of the 3D object to be reconstructed. Some of the edges may be redundant. Redundant edges imply here to the edges which are not required to define the object. They may be either some repeated edges or some extra edges generated. Redundant edges are classified into two types :

- overlapping edges.
- pathological edges.

3.3.1 Overlapping edge removal algorithm

Overlapping edges not only decreases efficiency of calculations but also causes computational complexities in forthcoming steps. A wireframe model cannot be said to be complete if it contains overlapping edges because this cannot be given as an input to solid model generation package. Solid model generation require generation of face loops and body loops. Presence of overlapping edges may result in formation of some invalid solid. Following steps are used to remove overlapping edges from probable 3D edge list generated in the previous step:

Step 1: Mark all the edges in *edge_list(3D)* as `overlapping`.

Step 2: Choose an `overlapping` edge e_i from *edge_list(3D)*, if all the `overlapping` edges are exhausted then **STOP**. Set $O_e \leftarrow e_i$, where O_e is set of all edges overlapping with e_i.

Step 3: For every edge $e_j \in$ *edge_list(3D)*, $e_i \neq e_j$ if e_j and e_i are collinear and e_j overlaps with one edge in O_e then set $O_e \leftarrow O_e \cup e_j$

Step 4: If there is only one edge in O_e, i.e no other edge in *edge_list(3D)* is overlapping with e_i, then mark this edge as non-overlapping and go to step 2.

Step 5: If there is more than one edge in O_e, remove all the edges $\in O_e$ from *edge_list(3D)* and sort all vertices of edges according to their co–ordinate value. Now let $v_1, v_2, v_3 .. ..v_n$ be the sorted vertex sequence, add $n-1$ edges (v_1, v_2), (v_2, v_3).. (v_{n-1}, v_n) to edge_list(3D) and mark them non overlapping. go to step 2.

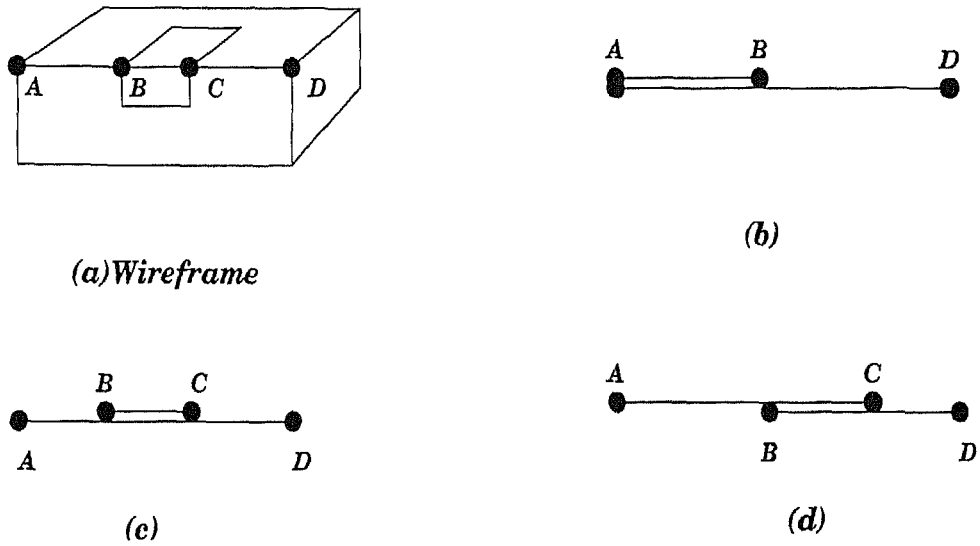


Figure 3.6: Overlapping edges in a wireframe

3.3.2 Pathological edge removal

At this stage all the overlapping edges are removed, but some other redundant edges, isolated edges and dangling edges may exist and before moving to next step they need to be removed. Algorithm discussed here deals with pathological cases that are redundant for generation of solid model. Let $N_a(V)$ be the number of adjacent edges at a vertex V, where $V \in$ *vertex_list*$(3D)$. Algorithm is as follows:

Step 1: Delete if there is any isolated vertex or an edge whose two vertices are same. This is $N_a(V) = 0$ shown in *Figure 3.7(a)*.

Step 2: If there is any dangling edge i.e. $N_a(V) = 1$, remove this edge from *edge_list(3D)* and vertex V from *vertex_list(3D)* . Shown in *Figure 3.7(b)* edge e and vertex highlighted by circle are removed from *edge_list(3D)* and *vertex_list(3D)* respectively.

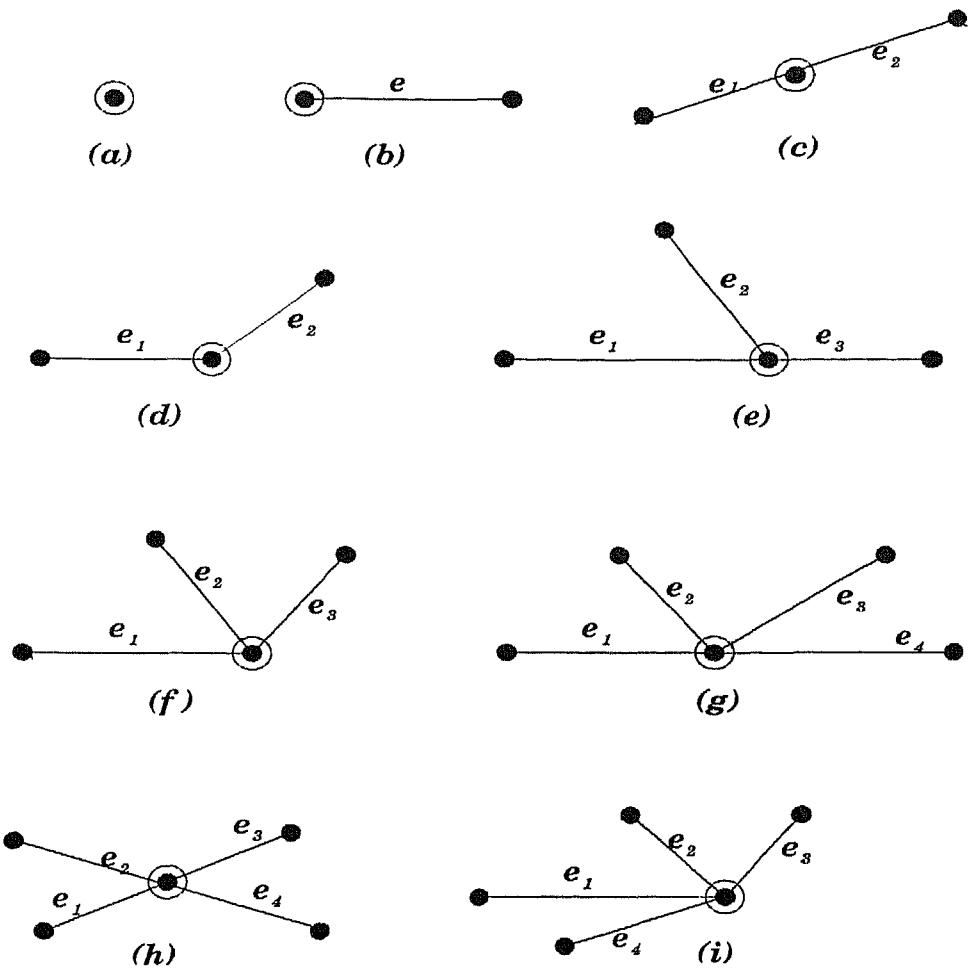


Figure 3.7: Pathological edges that needs to be handled; edges shown are removed from *3D edge list* and vertices highlighted by circle are removed from *3D vertex list.*

Step 3: If $N_a(V) = 2$ and two edges are collinear remove this vertex V and merge two edges at this vertex. Shown in Figure 3.7(c) edges e_1 and e_2 are merged at highlighted vertex and this vertex is removed from *vertex_list(3D).* If two edges are not collinear remove the two edges from *edge_list(3D)* and vertex from *vertex_list(3D).* Shown in *Figure 3.7(d)* both e_1 and e_2 are removed from *edge_list(3D)* and vertex V is removed *vertex_list(3D).*

Step 4: If $N_a(V) = 3$ and two of the three edges are collinear remove this vertex and third edge, merge two collinear edges at this vertex. Shown in *Figure 3.7(e),* e_2 is removed and e_1, e_3 are merged at highlighted vertex.

Step 5: If $N_a(V) = 3$, none of the two edges are collinear, but all three edges are

co–planar delete this vertex and all three edges. With reference to *Figure 3.7(f)* e_1, e_2 and e_3 are removed from *edge_list(3D)*, highlighted vertex is removed from *vertex_list(3D)*.

Step 6: If $N_a(V) \geq 4$ if all the edges are co–planar and only two edges are collinear as shown in *Figure 3.7(f)* merge two collinear edges at this vertex and remove this vertex and edges other than two collinear edges.

Step 7: If $N_a(V) \geq 4$ all the edges co–planar and two pair of edges are collinear or none of the edges are collinear remove this vertex from *vertex_list(3D)* and all edges from *edge_list(3D)* as shown in *Figure 3.7(g) and (h).*

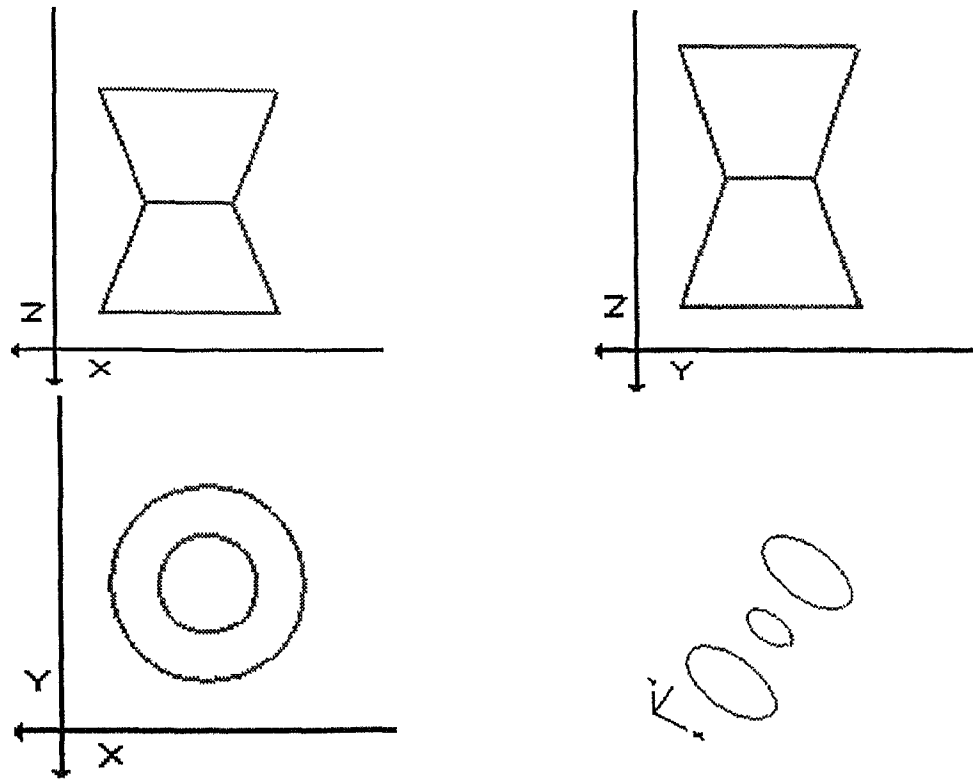


Figure 3.8: Example of output generated after third step

3.4 Curved surface representation

The output after redundant edges removal step will consist of valid 3D edges. But there is no information about curved surfaces. The output will look like Figure 3.8 for the given views shown. So we are having information about the curved edges only,

but we do not know the connectivity of these arcs. There is no information about the joining surface of these arcs. In this section the algorithm for representing surface will be presented. The algorithm is based on back–projection. The two arcs are connected to each other in three views respectively. They form a surface and the nature of surface is determined by the connectivity between them. Algorithm is as follows:

Step 1: Select an arc from list of generated 3D arcs a_i. If all the arcs in arc list are exhausted then go to step 8. Let projection of the arc in three orthographic planes be fe_i, te_i, se_i respectively.

Step 2: If projection of this arc in each of the three views is circular arc then this arc represent a spherical surface because only spherical surface can be projected as circular arc in all three views. Mark this arc as spherical surface.

Step 3: Select another edge from 3D arc list a_j, $a_i \neq a_j$. Let fe_j, te_j, se_j be its projection in front, top and side views respectively.

Step 4: If either fe_i and fe_j are same or there exist an edge $e \in e\_list(f)$ such that it connects the edges fe_i and fe_j, then go to next step. Otherwise go to Step 1.

Step 5: If either te_i and te_j are same or there exist an edge $e \in e\_list(t)$ such that it connects the edges te_i and te_j then go to next step, otherwise go to Step 1.

Step 6: If either se_i and se_j are same or there exist an edge $e \in e\_list(s)$ such that it connects the edges se_i and se_j then these two arcs form a surface. Mark the two arcs as connected, else go to Step 1.

Step 7: If the two arcs are connected by line in views then the type of surface connecting the two is cylindrical or conical, and if they are connected by arcs then the surface connecting the two is torodial. Save this surface information and go to step 1.

Step 8: Facet all the surfaces, in this step we break the surface into rectangular patches. All the auxiliary lines generated to facet the surface are then added to the e_list(3D) and all the arcs are removed from e_list(3D). Now wireframe is represented by the lines only.

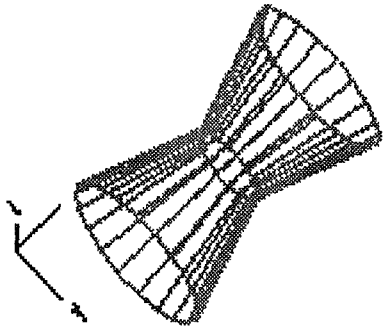


Figure 3.9: Example of output generated

The output now contain wireframe model of the object which was described by three orthographic projections. The surface information at the end is in faceted B–rep format but user can obtain exact B-rep model. This output can now be given to a programme which can generate solid model from wireframe model.

Chapter 4

Result and Analysis

The algorithm discussed was implemented in C++ and for graphic display **OpenGL** was used. Result shown here are graphical output produced by the software. The software is tested for some of objects which are presented here. The object presented here explore the wide variety of cases which can be handled by software developed.

Figures4.1—4.7 show the results for polyhedral objects, Figures4.7—4.18 show objects with cylindrical and conical surfaces. Figures4.19—4.21 show spherical surfaced object reconstructed. The processing time taken is also shown and it can be seen that processing time taken for reconstruction is very small, even for complex part shown in Figure4.2. processing time taken is 0.05 seconds.

Objects shown in Figures4.1—4.21 deal with a large variety of cases involving holes, handles, blind holes, planar, cylindrical,conical and spherical surfaces. The objects which could not be dealt were those in which cylindrical and spherical, cylindrical and cylindrical or spherical and spherical surfaces intersect. These cases form edges which are neither circular arc nor a straight edge but the intersection edge is some complex 3D curve.

Specification for the computational platform :

Computer specification	IBM NetVista
Microprocessor	Pentium–III
RAM	64MB
Operating system	Linux
Graphics used	OpenGL libraries

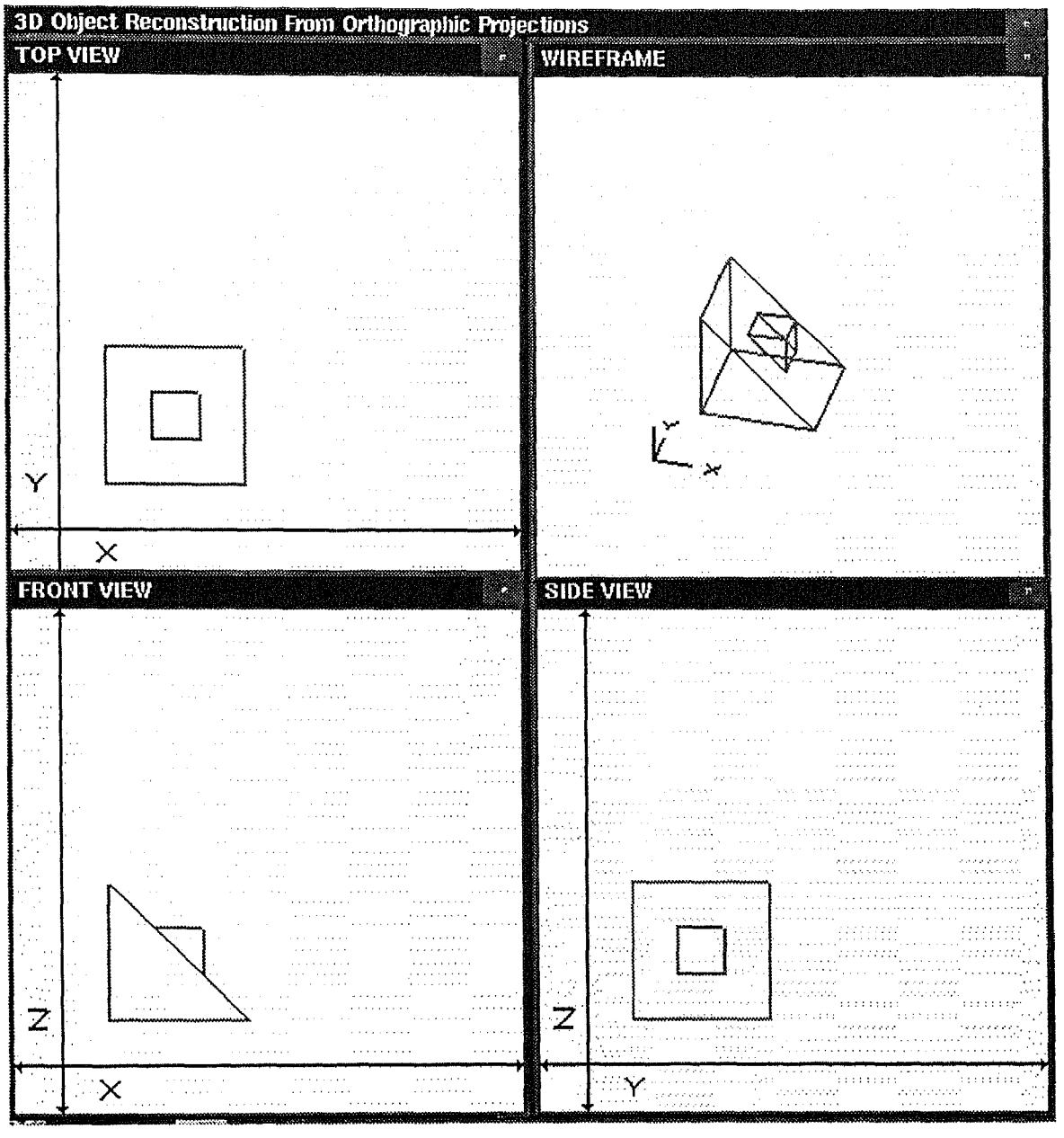


Figure 4.1: Two wedges. Processing time = 0.01Sec.

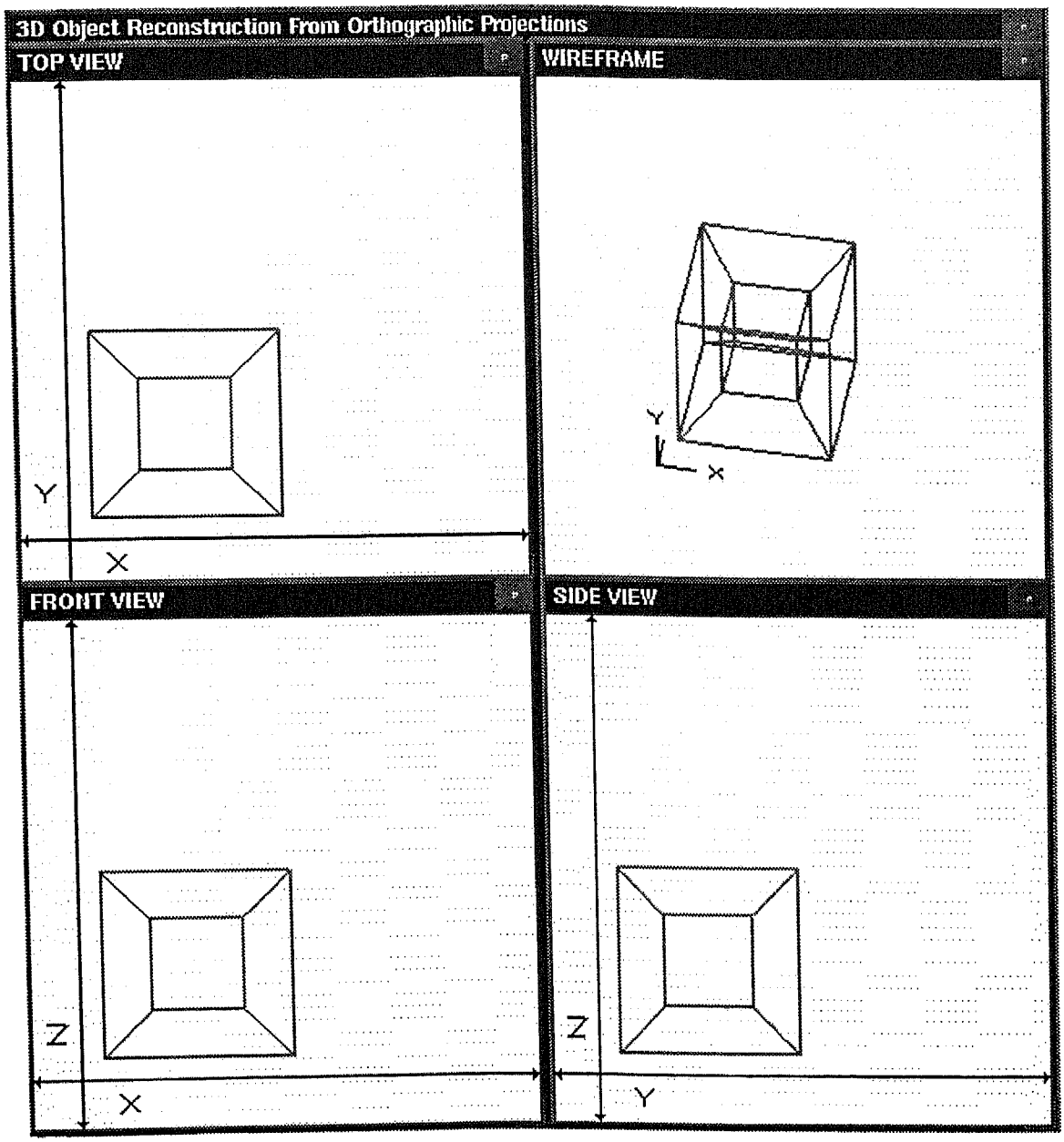


Figure 4.3: Cuboid, an ambiguous Polyhedral object. Processing time = 0.01Sec.

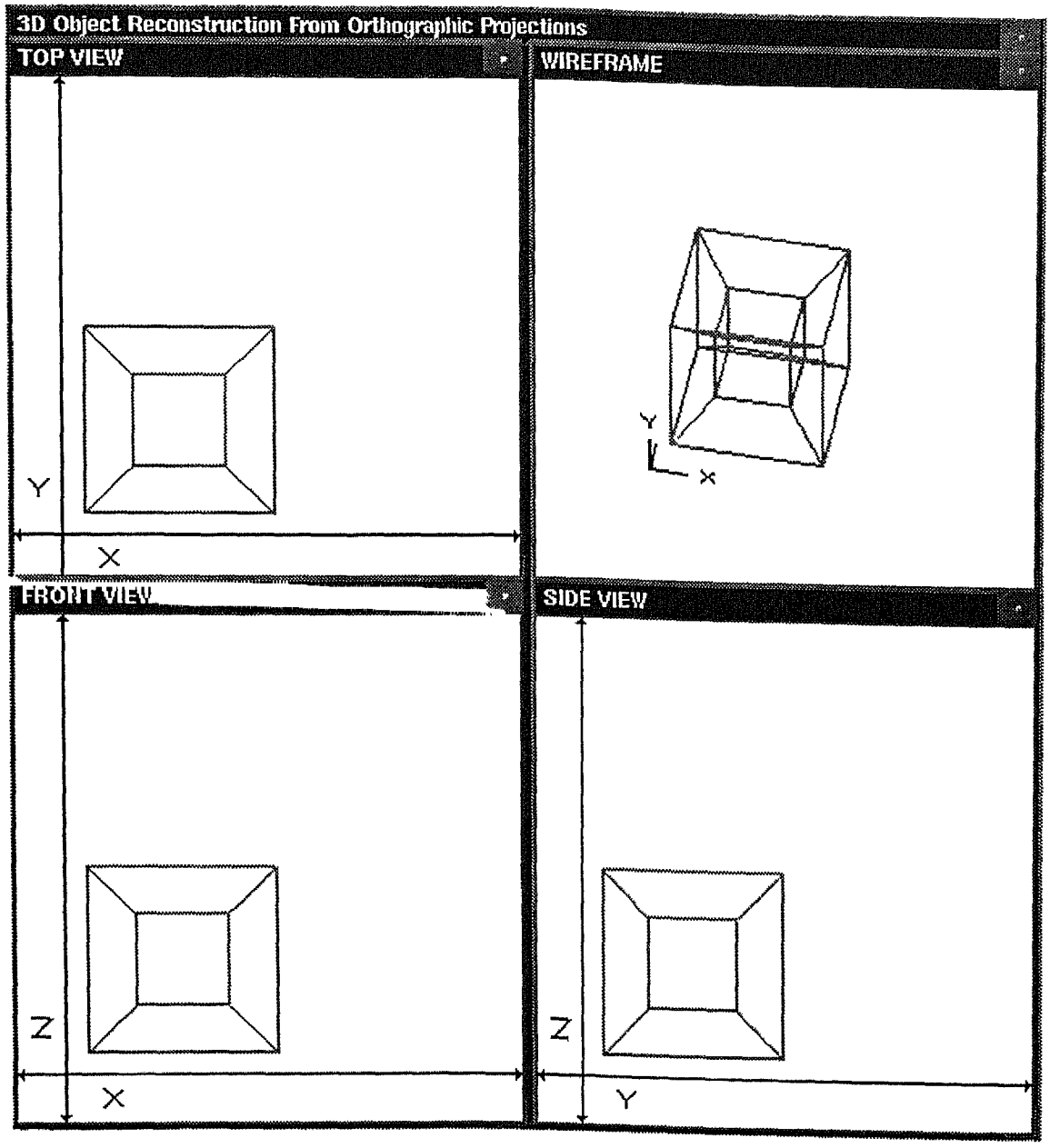


Figure 4.3: Cuboid, an ambiguous Polyhedral object. Processing time = 0.01Sec.

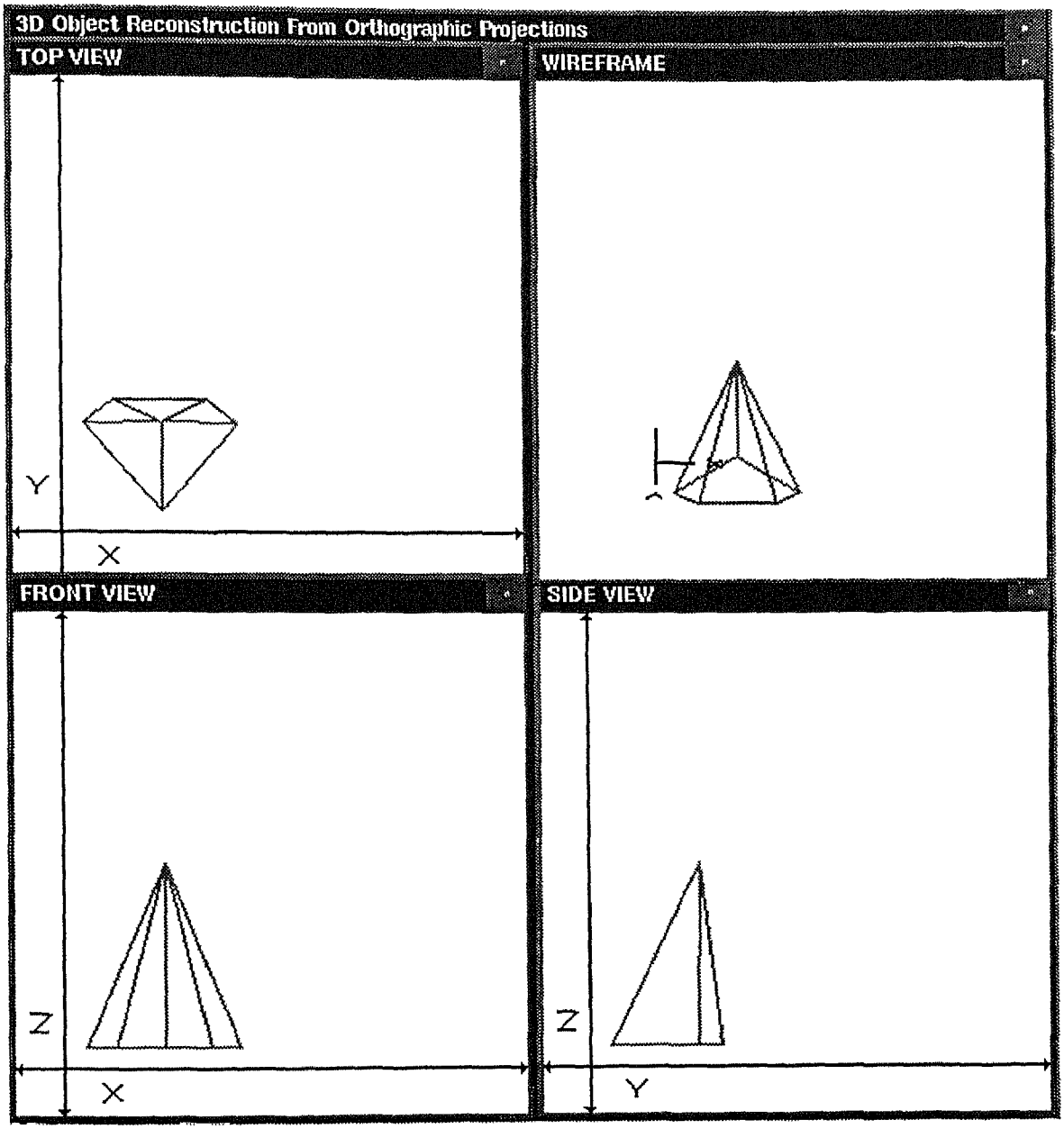


Figure 4.4: Pyramid, A non–uniform polyhedral object. Processing time = 0.02Sec.

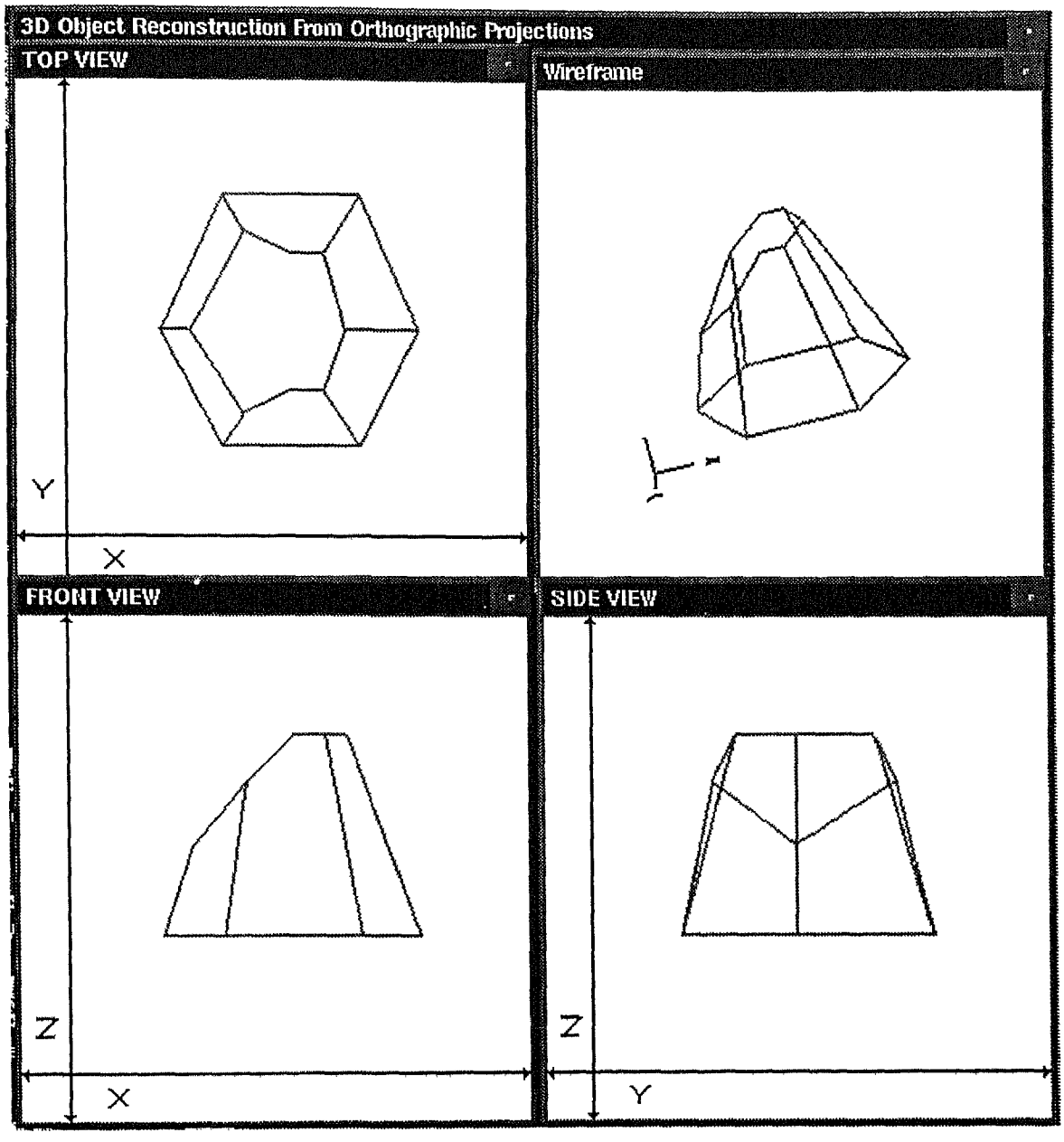


Figure 4.5: A truncated Pyramid. Processing time = 0.03Sec.

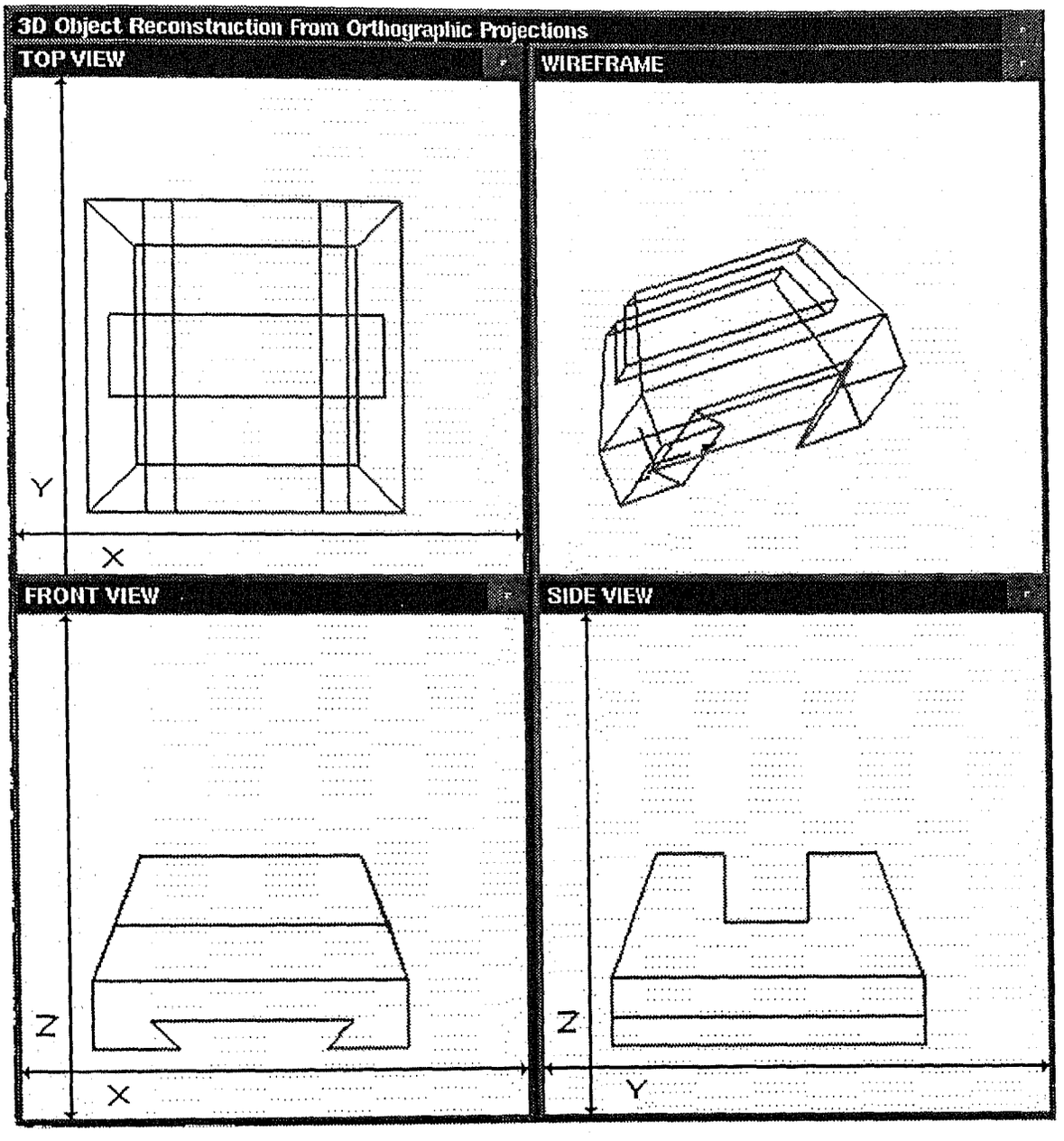


Figure 4.6: A complex engineering polyhedral object. Processing time = 0.03Sec.

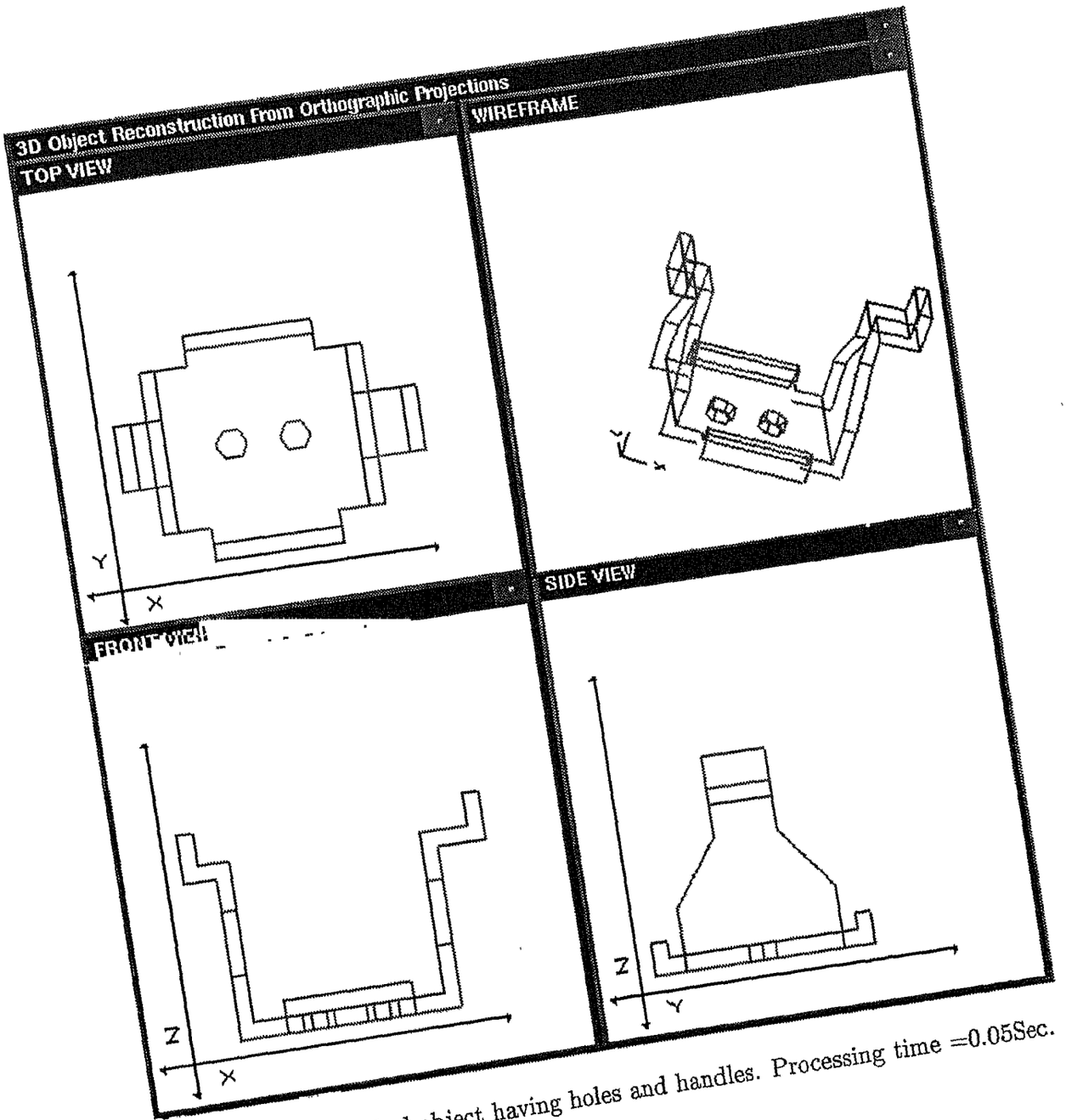


Figure 4.7: Polyhedral object having holes and handles. Processing time =0.05Sec.

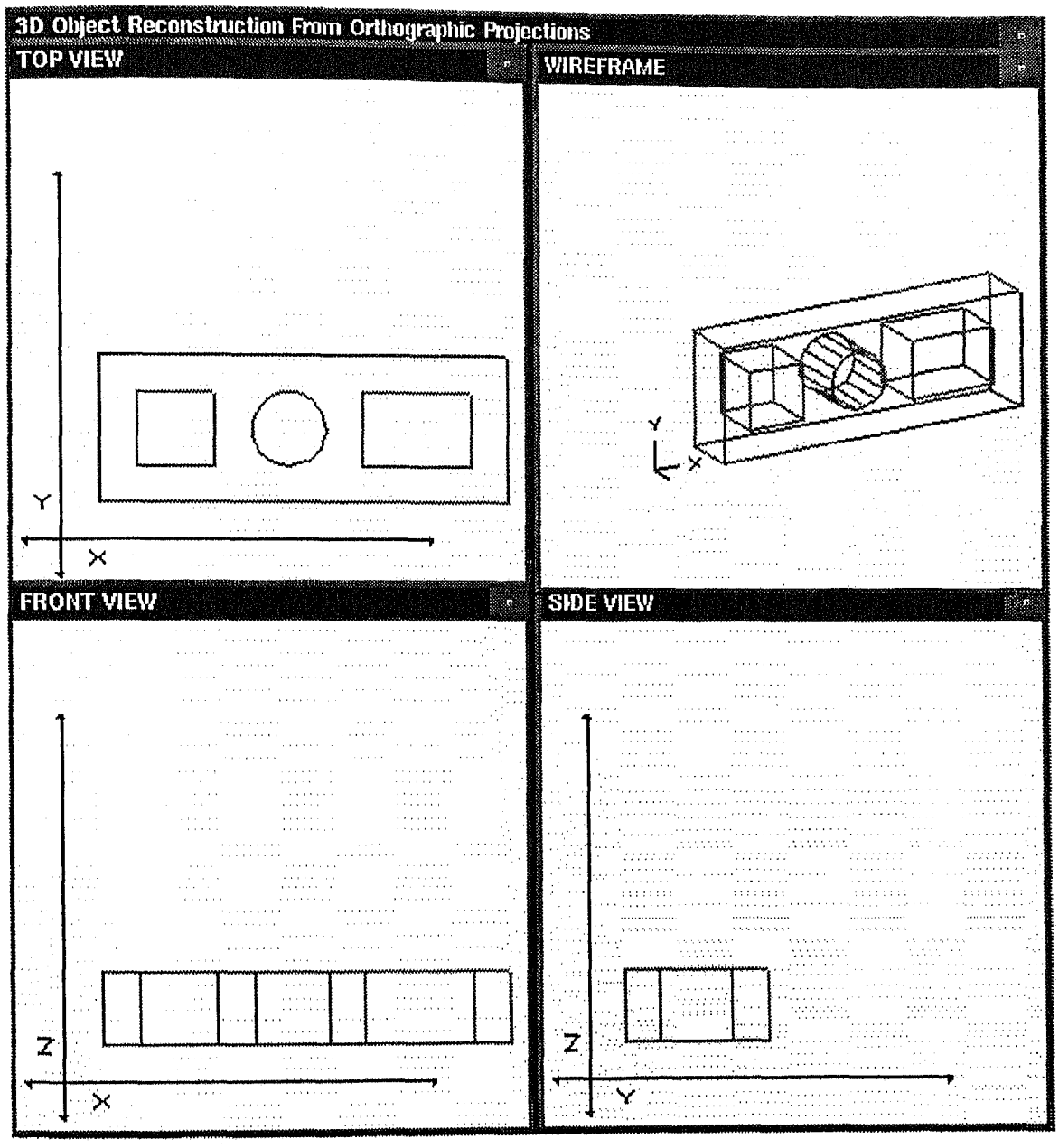


Figure 4.8: Object having both cylindrical and rectangular through holes. Processing time =0.01Sec.

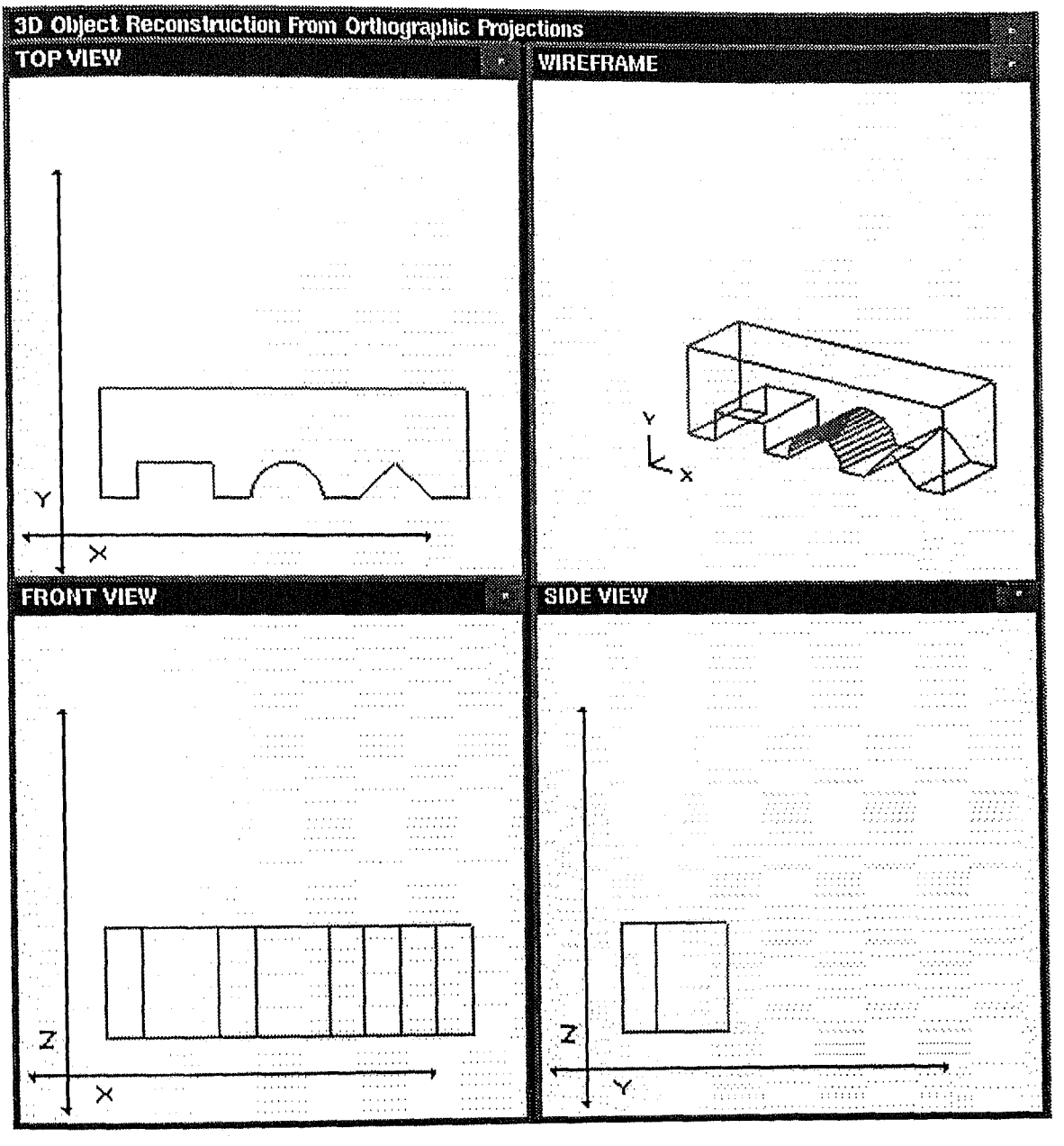


Figure 4.9: Object having triangular, rectangular and circular slots. Processing time = 0.01Sec.

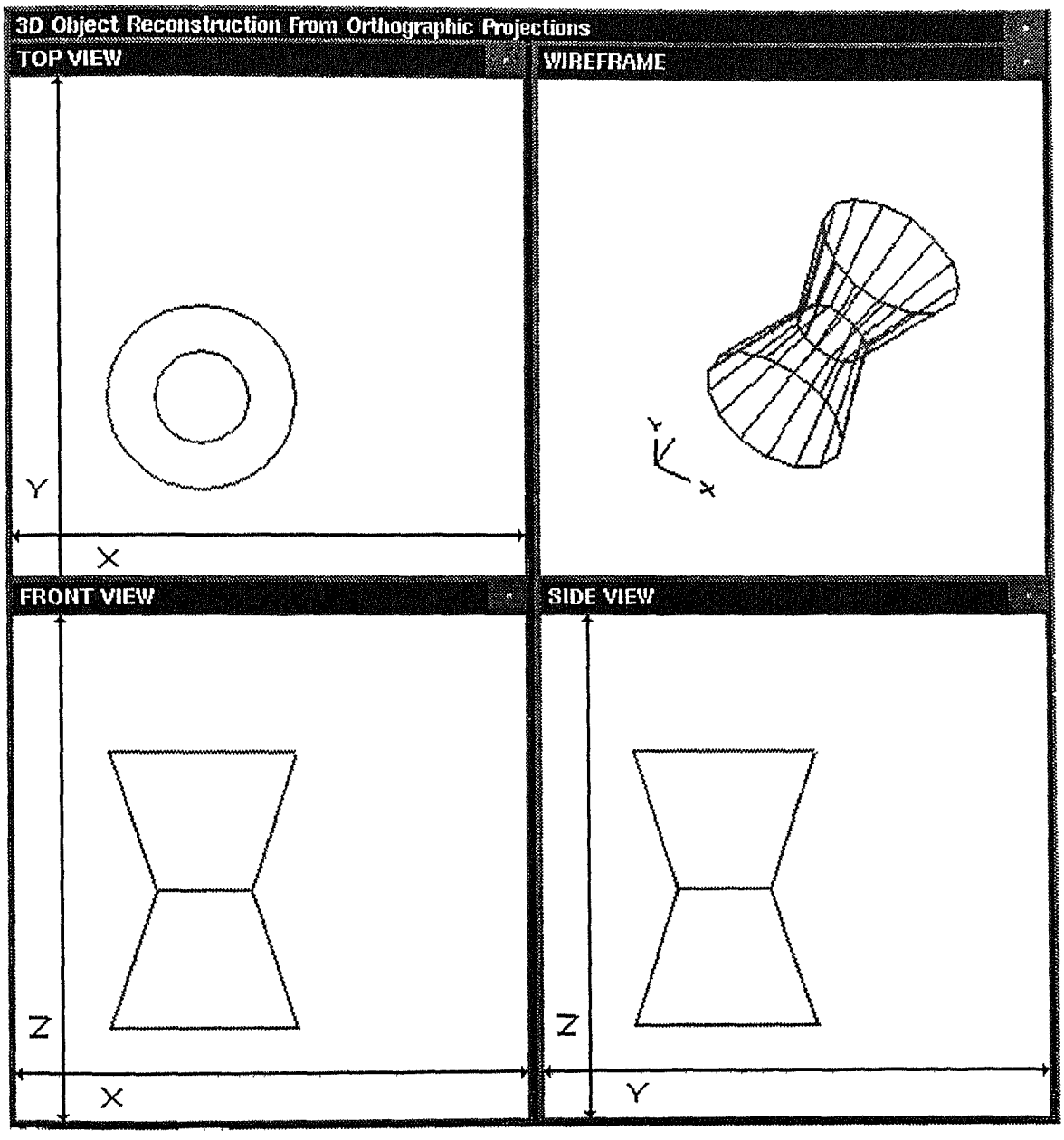


Figure 4.10: Object with double cylindrical surface. Processing time = 0.01Sec

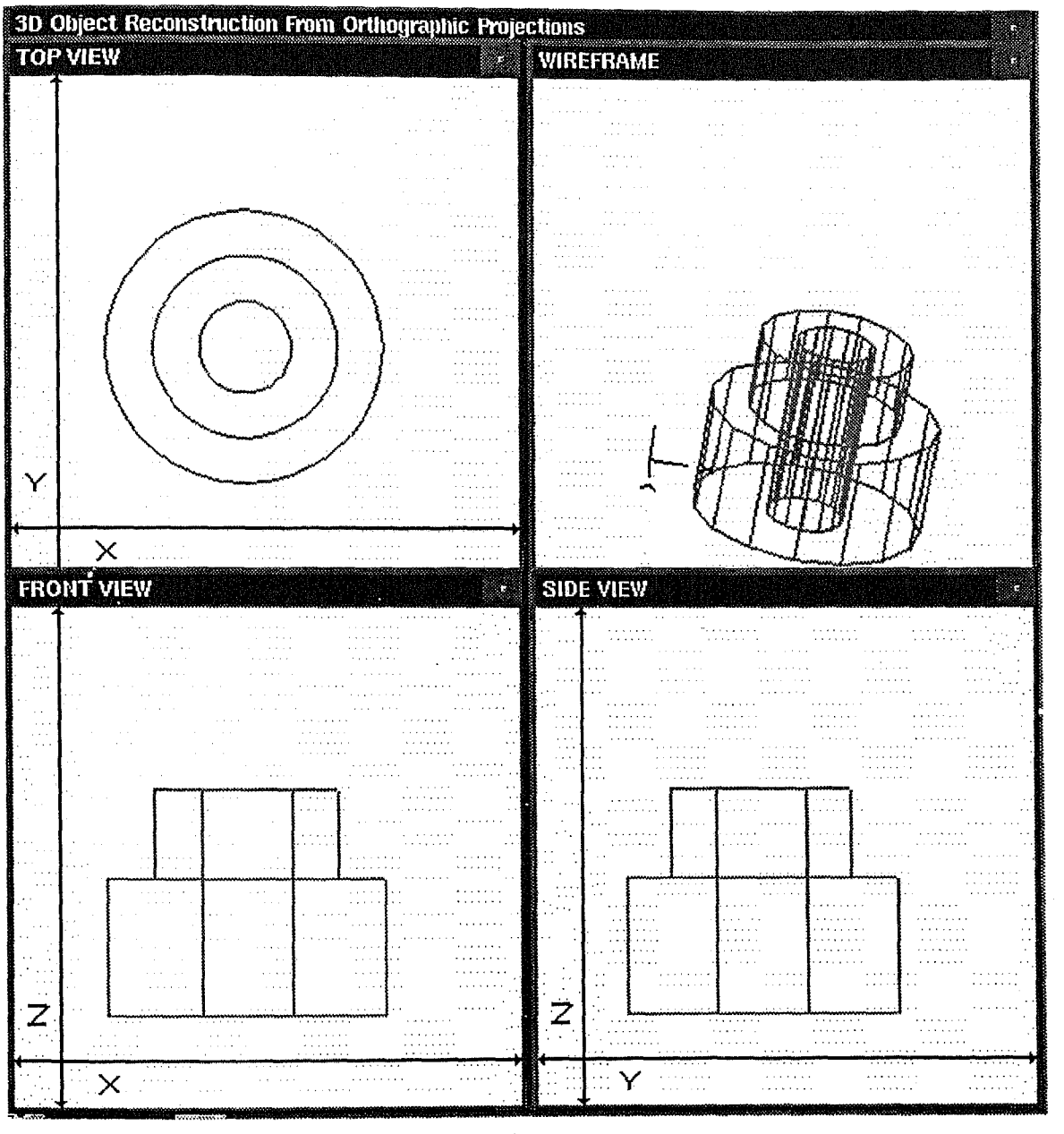


Figure 4.11: Stepped Cylinder with a cylindrical hole. Processing time = 0.01Sec

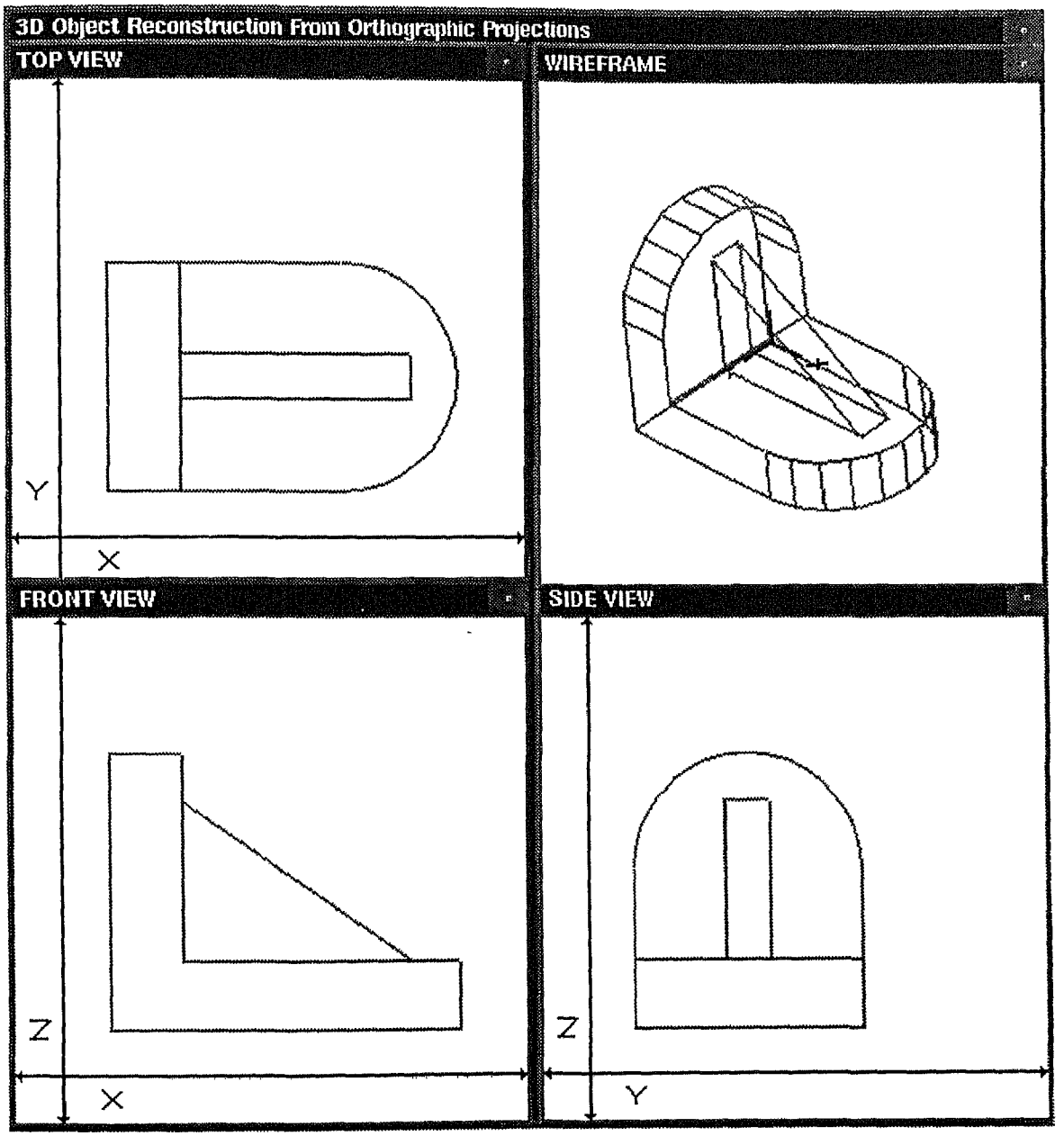


Figure 4.12: Bracket with rib. Processing time = 0.01Sec.

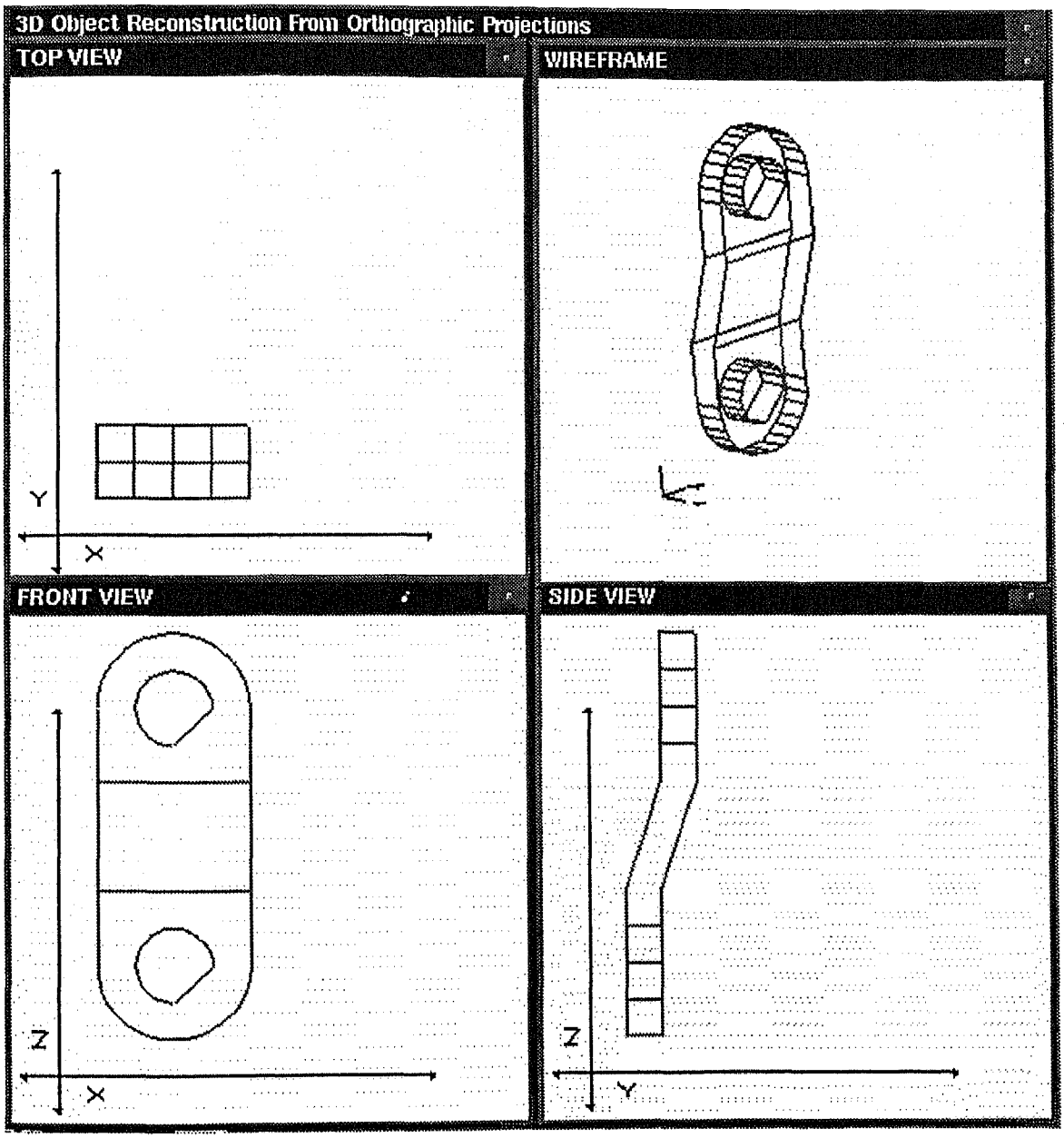


Figure 4.13: An object with holes that are not completely circular. Processing time = 0.01Sec.

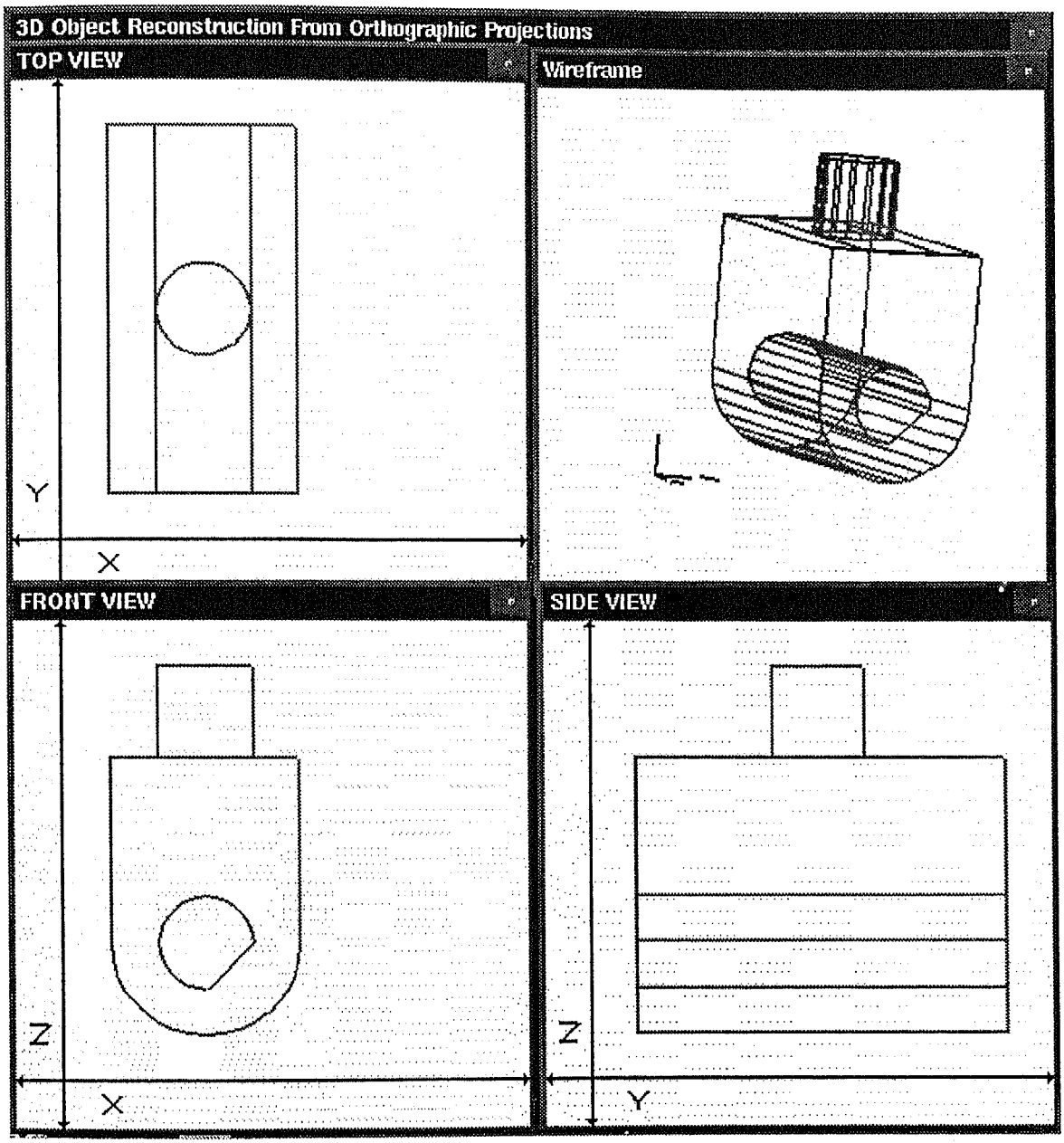


Figure 4.14: Another example of cylindrical surfaces. Processing time = 0.02sec.

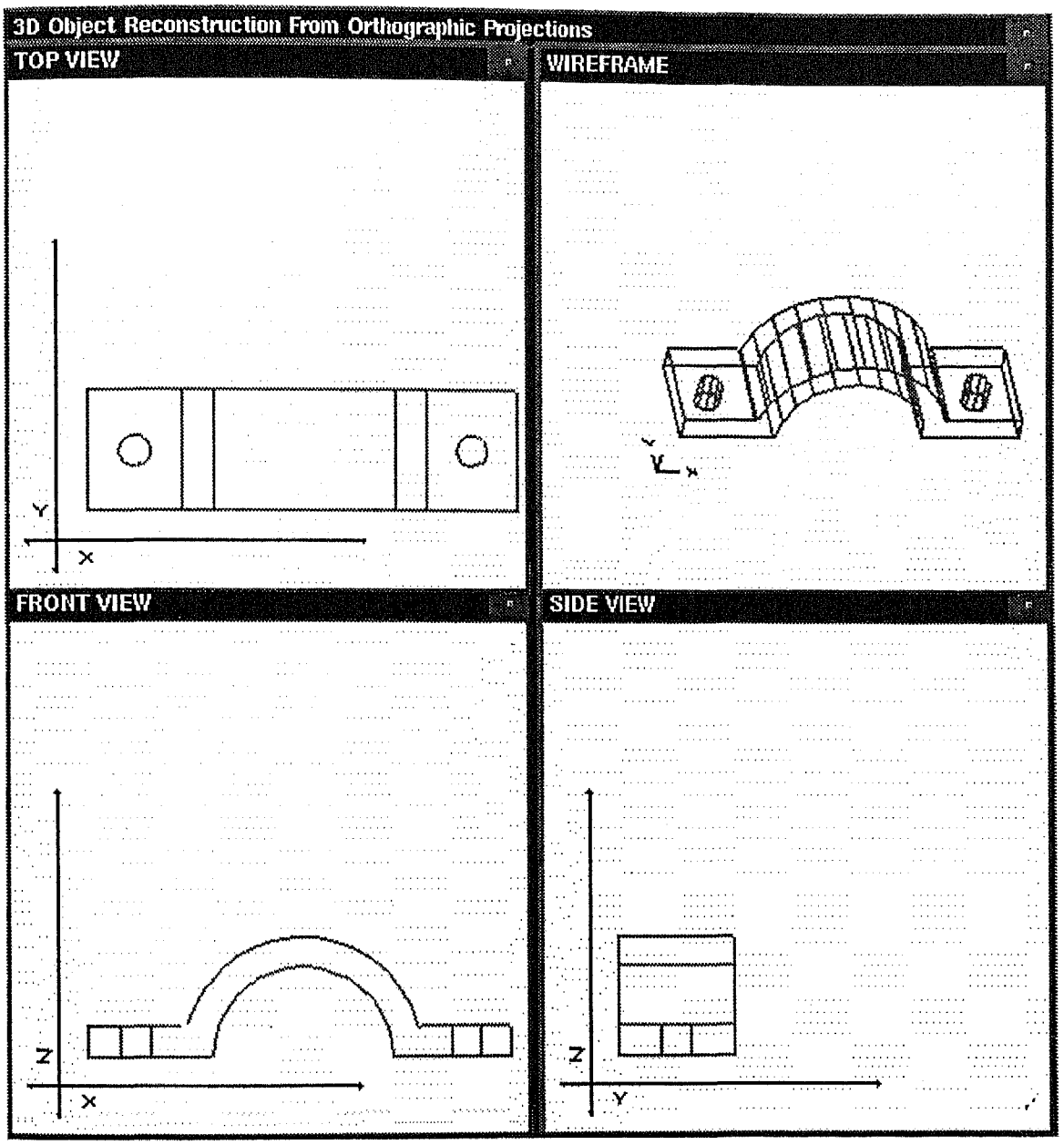


Figure 4.15: An example of half cylindrical surface. Processing time = 0.01Sec

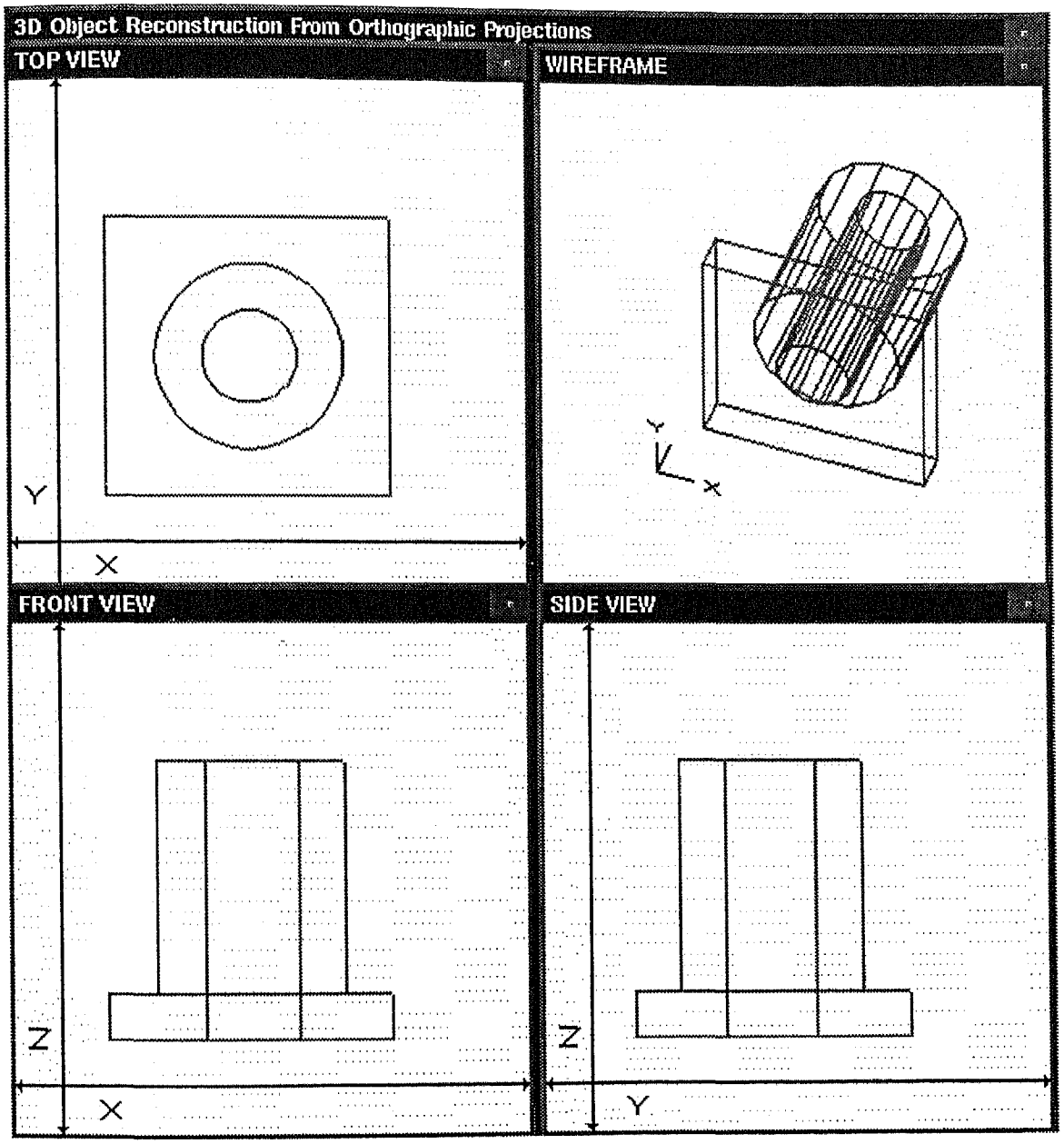


Figure 4.16: Processing time taken = 0.01 Sec

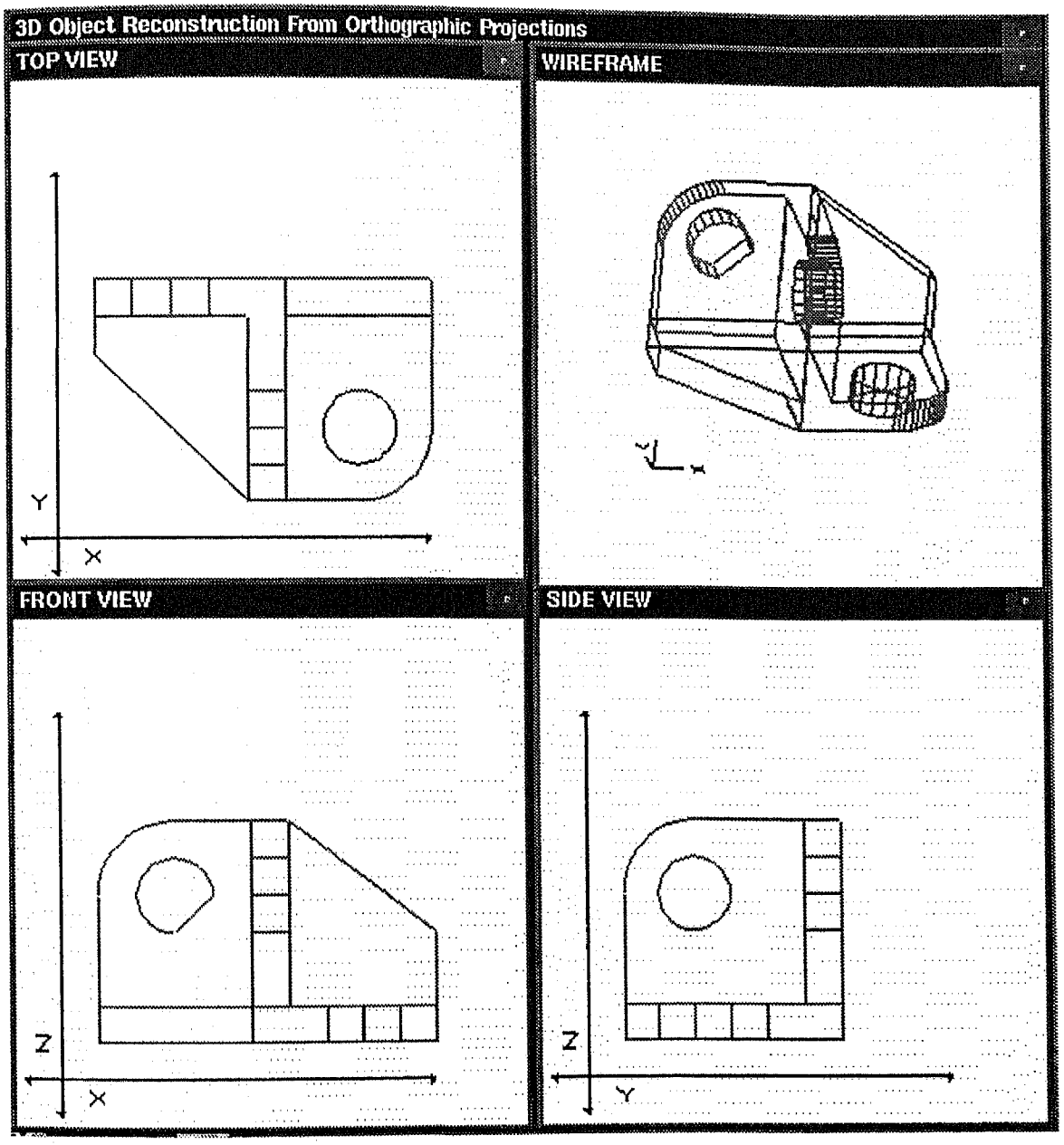


Figure 4.17: A complex object with 6 cylindrical surfaces, ribs and holes. Processing time = 0.02Sec

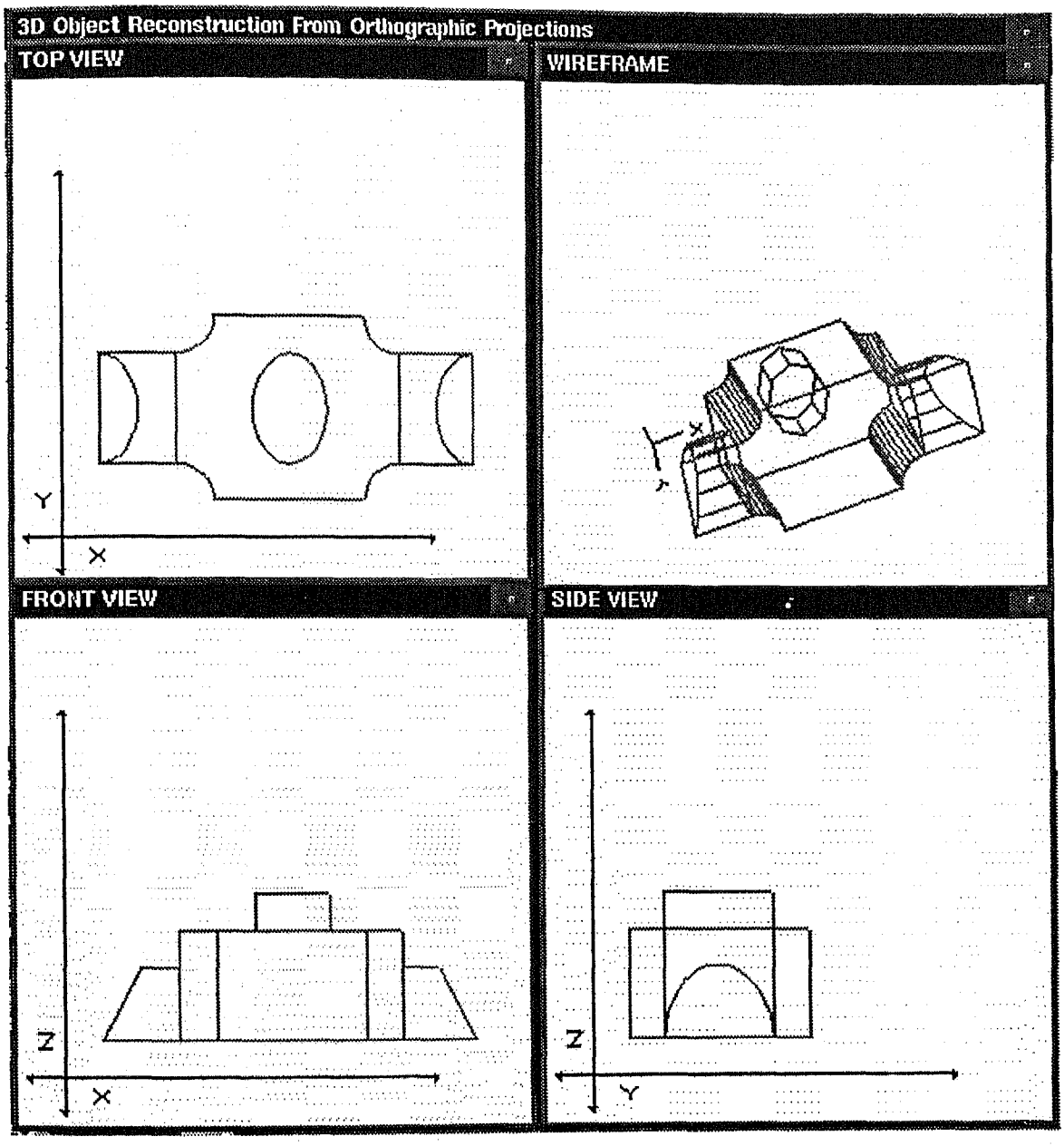


Figure 4.18: An object having circular faces which are not parallel to any axis. Processing time = 0.02Sec

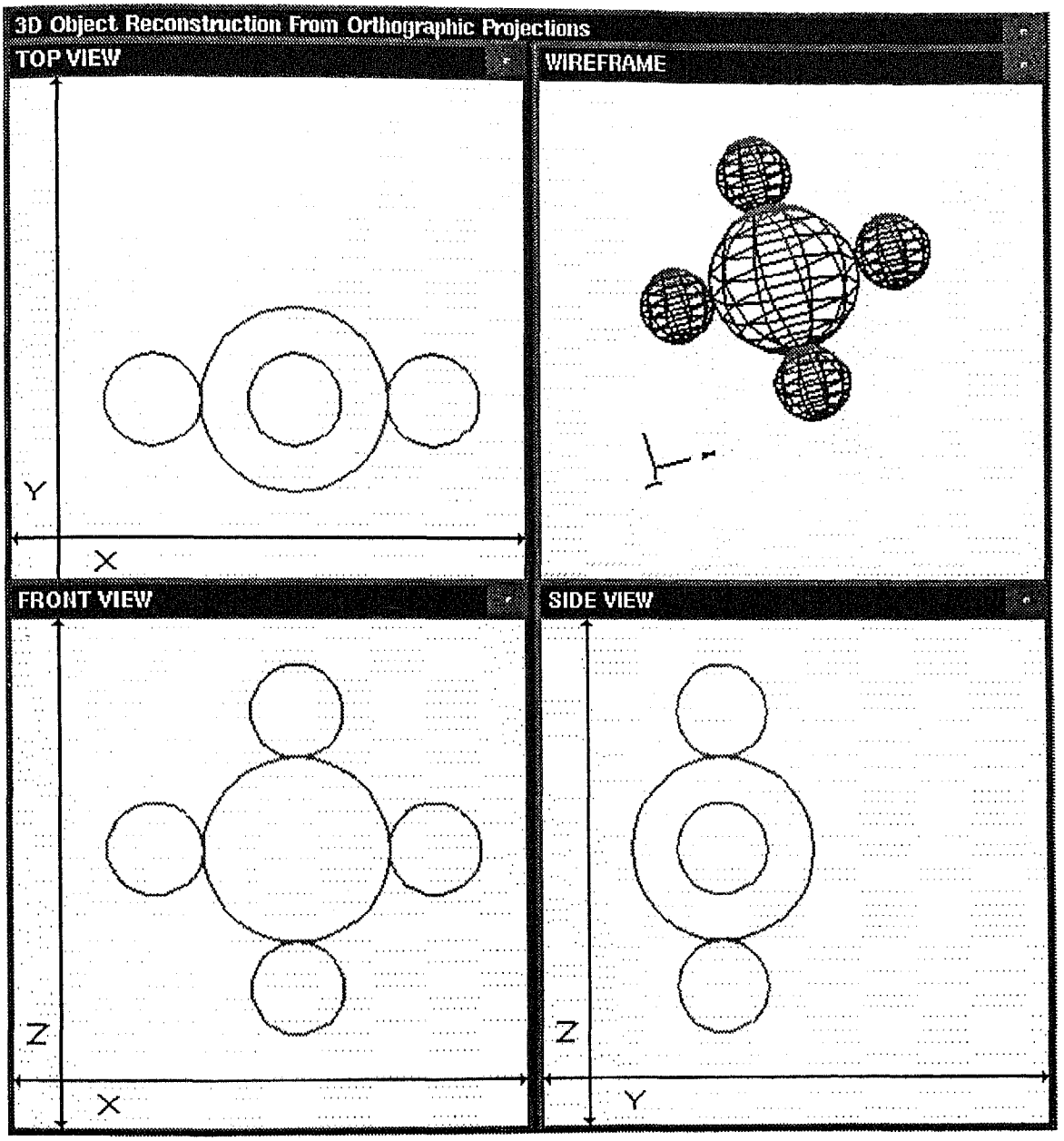


Figure 4.20: Four spheres. Processing time = 0.01Sec.

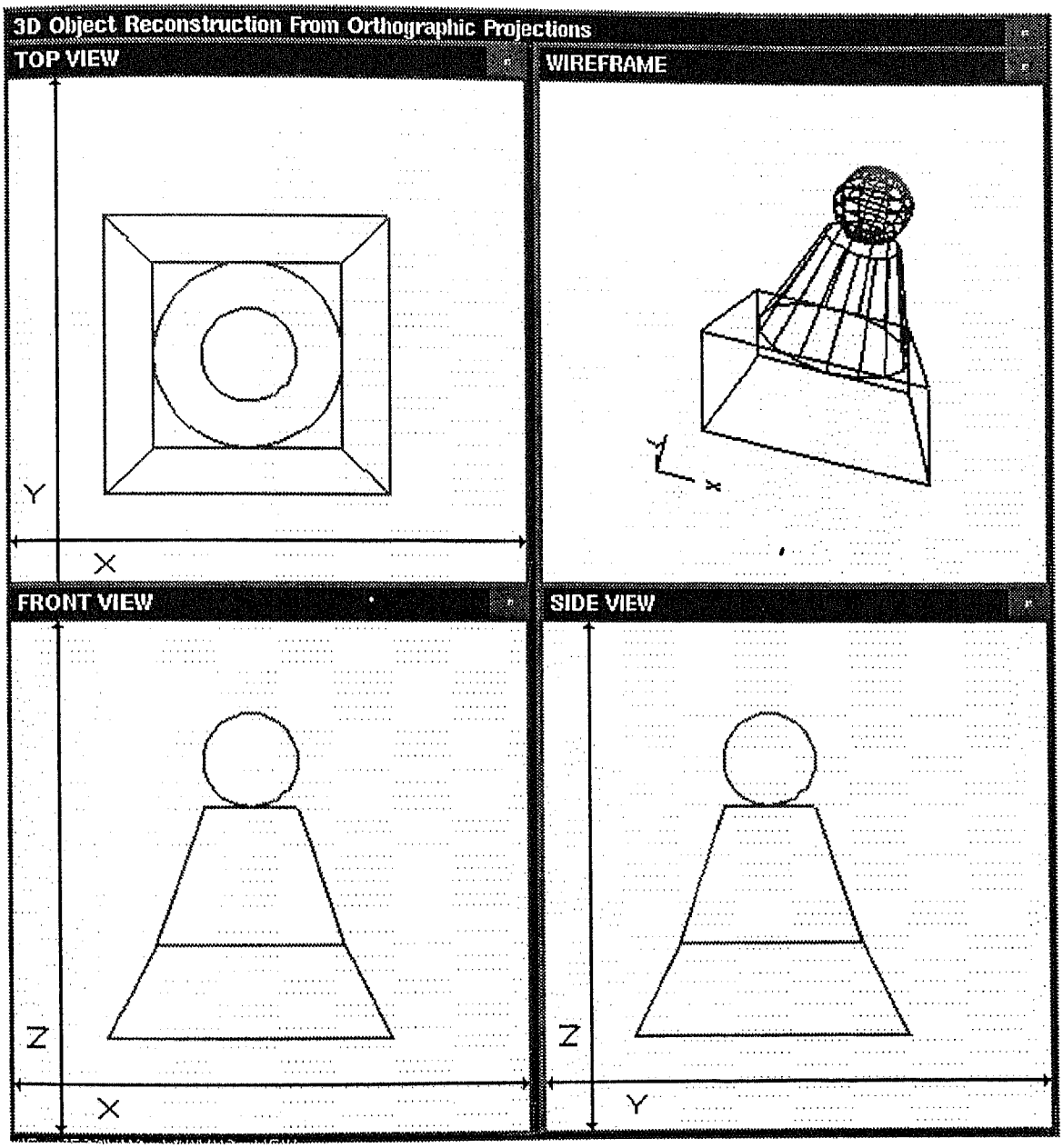


Figure 4.21: Object having Planar, cylindrical, spherical surfaces. Processing time = 0.01Sec.

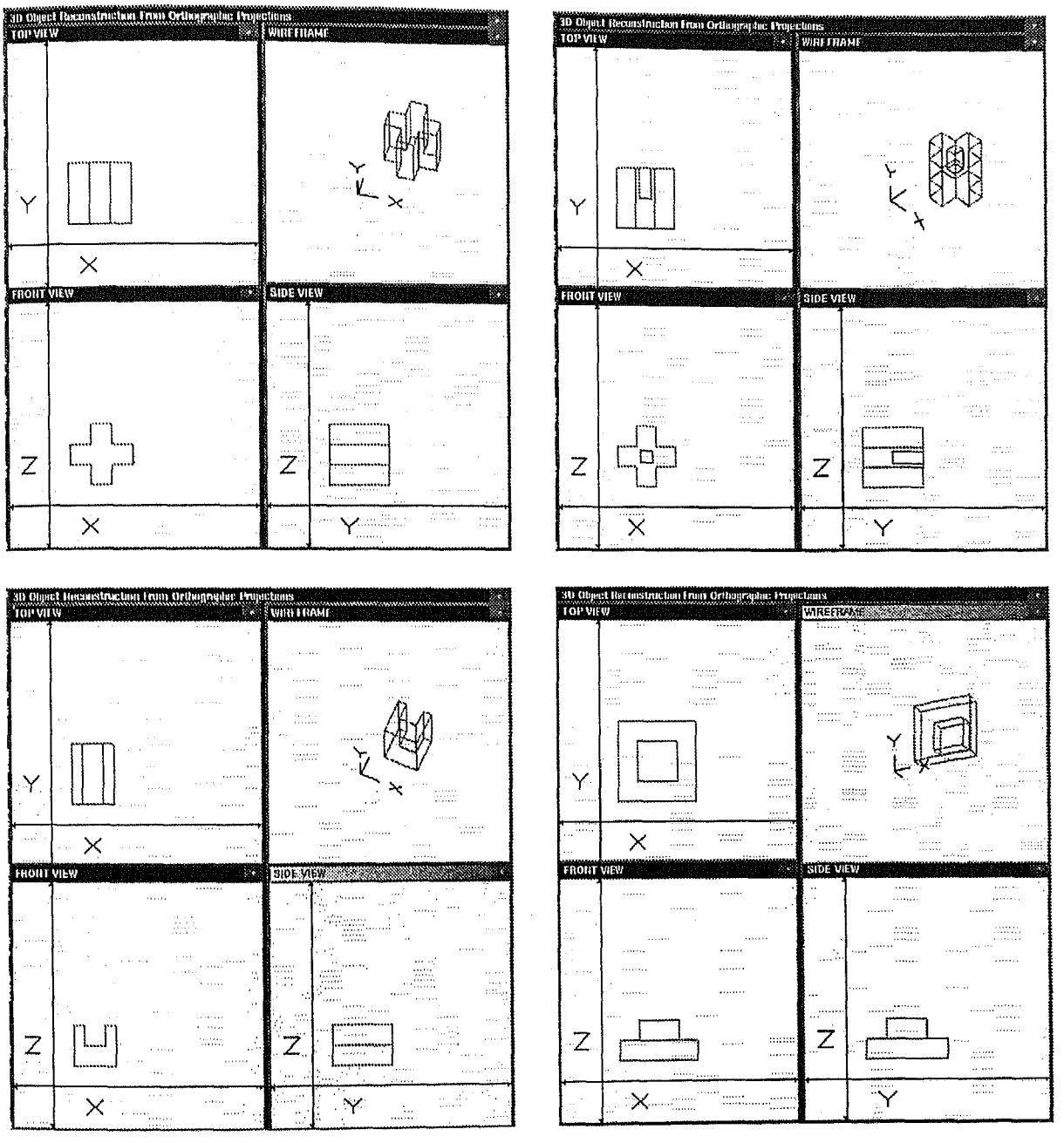


Figure 4.22: Some more Examples of polyhedral Objects generated

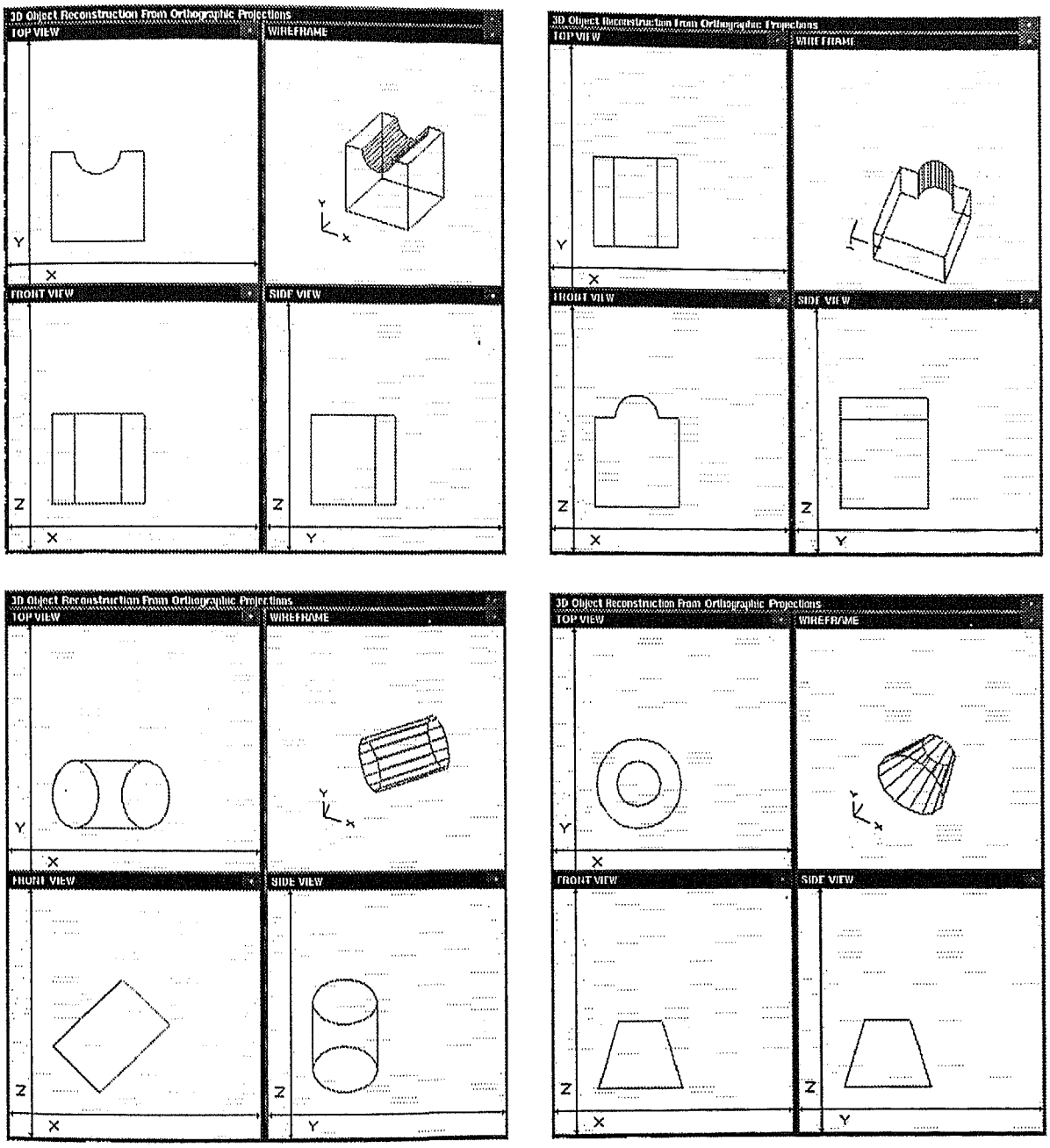


Figure 4.23: Some more Examples of cylindrical objects

Chapter 5

Conclusion and scope for future work

Here we have developed and implemented an algorithm to generate the wireframe object from given three projections efficiently. Polyhedral objects as well as object having circular or elliptical edges have been handled successfully. The work present is an computationally efficient algorithm for generation of wireframe object from three views given. By classification of edges in the pre–processing step we need not compare every edge in one view with all edges in other view but by propagating constraints, the number of comparisons required are minimized. Sorting of data in pre-processing step was also helpful in reducing the search for matching edges.

5.1 Achievements

1. Algorithm perform satisfactorily for spherical, cylindrical and conical objects apart from polyhedral objects.
2. The generated wireframe model is complete in itself i. e. it does not contain any redundant edge, as all the redundant edges are removed during step three.
3. Checking of edges also removes edges which may be generated due to wrong input so it can avoid some "noise" in input.
4. Output for surfaces can be generated in exact and faceted B-rep, which is convinient input to solid model generation algorithm.
5. Preprocessing makes life easier for user as he/she is not required to mention each edge explicitly.

5.2 Limitations

1. If there are some missing edges in input or input is incomplete then the algorithm can not fill the "void" created due to these absent edges Input has to be complete. The lines which can be ignored by a trained person while reading projections can not be ignored while giving input to software developed here

2. Although it can deal with cylindrical, conical and spherical surfaces, the complex 3D curved edge formed by intersection of two curved surcfaces can not be reconstructed.

3. Here we have generated wireframe model which itself has a lot of limitation and limited applications. So this work will be incomplete until a solid model generation part is added to it.

4. It cannot understand some special engineering symbols such as threads, fillet etc.

5. Only three projections can be used. Sectional view or auxilliary view can not be used to provide some extra information which otherwise is very difficult to represent in engineering drawing.

6. It require three views to reconstruct wireframe even for those objects which can be interpreted by two views only.

Inspite of these limitation the work presented here is a starting step in the direction of a new automated approach to generate 3D objects. Continuation of this work may lead to a completely automatic solid modeller and can proved to be helpful in saving of skilled labour time.

5.3 Scope for future Work

1. This work is limited to cylindrical, conical and spherical surface. So this can be extended for free form surfaces.

2. This algorithm can not identify mechanical engineering symbols such as fillet, threads etc. So an algorithm can be added to this which can identify these special symbols in three views and generate it in final three dimensional object.

3. The software generates a wireframe model. A solid model conversion routine has to be added to it, so as make this software complete. *Markowasky and Wesley*[21,22] have discussed an algorithm to generate all possible solid objects from given wireframe. *Yen et. al.*[9] has also proposed an algorithm for the same.

4. Some engineering drawing can be interpreted from two views only. So this feature also can be added to present system. A system than can be established which will generate 3D object from two views if unique solutions exists, otherwise it should generate all possible solutions.

5. Since engineering drawings are prepared by humans, human error factor can not be tolerated. The input may not be exact, *i.e.* some edges might be missing in input or some wrong edges might be present. The work dealt here requires exact input and may generate some nonsense object if input itself is not complete. It is not possible to take into consideration all the types of human errors but some very common mistakes can be taken care for.

6. This work does not treat hidden lines separately. Hidden lines if taken care for will be helpful in reducing the computational complexity of algorithm.

7. Sectional and auxilliary views are used to represent some special features of object. If an algorithm can be developed to utilize their information then a lot of engineering drawings can be explored.

8. The ultimate objective is to construct 3D solid model from the scanned engineering drawings prepared by draftsman. If an image processing software is added to this then it will be possible to take input as scanned image.

Bibliography

[1] Shum, S. S. P., Lau, W. S., Yuen, M. M. F. and Yu, K.M. (2000). Solid reconstruction from Orthographic Views using 2-Stage Extrusion. *Computer-Aided-Design* Vol. 33, pp 91–102

[2] Mukarjee, A., Sasmal, N. and Sastry, D. S. (1999). Efficient Categorization of 3D Edges from 2D Projectins. GREC'99,LNCS pp 288-297.

[3] Tanaka, M., Iwama, K., Hosoda, A. and Watanabe, T. (1998). Decomposition of a 2D Assembly Drawing into a 3D part Drawings. *Computer-Aided-Design* Vol. 30, No. 1, pp37—46

[4] Masuda, H. and Numao, M. (1997) A Cell-Based Approach for Generating Solid Objects from Orthographic Projections. *Computer-Aided-Design* Vol. 29, No. 3, pp 177—187

[5] Kuo, M. H. (1997). Reconstruction of Quadric Surface Solids From Three-View Engineering Drawings. *Computer-Aided-Design* Vol. 30, No. 7, pp 517–527

[6] Shpitalni, M. and Lipson, H. (1996). Identification of Faces in a 2D Line Drawing Projectoion of a Wireframe Object. *IEEE Trans. on Pattern Analysis and Machine intelligence* Vol. 18, No. 10, pp 1000—1012

[7] Tombre, K. and Ah-Soon, C. (1995). A step Towards Reconstruction of 3D CAD Models from Engineering Drawings. *ICDAR'95*

[8] Tombre, K. and Dori, D. (1995). From Engineering Drawings to 3D CAD Models: Are we Ready Now? *Computer-Aided-Design* Vol. 27, No. 4, pp 243—255

[9] Yan, Q., Chen, C. L. P. and Tang, Z. (1994) Efficient Algorithm for tne Reconstruction of 3D Objects from Orthographic Projections. *Computer-Aıded-Design* Vol. 26, No. 9, pp 699—717

[10] Shin, B. S. and Shin Y. G. (1994). Fast 3D Model Reconstruction from Orthographic Views *Computer-Aıded-Desıgn*

[11] Meeran, S. and Pratt M. J. Automatic Feature Recognition from 2D Drawings. (1993). *Computer-Aided-Design* Vol. 22, No. 5, pp 7-17

[12] Dori, D. (1989) A Syntatic/Geometric Approach to Dimensions in Engineering Machine Drawings. *Computer Vısıon, Graphıcs and Image Processıng* Vol. 47, No. 3, pp 271—291

[13] Gujar, U. and Nagendra, I. (1989) Construction of 3D objects from Orthographic Views. *Computer Graphics* Vol. 13, No. 4, pp 505—521

[14] Chen, Z. and Perng, D. B. (1988) Automatic Reconstruction of 3D Solid Objects from 2D Orthographic Views. *Pattern Recognition* Vol. 21, No. 5, pp 439—449

[15] Bin, H. (1986). Inputting Constructive Geometry Represantations DirectlyZ from Orthographic Views. *Computer-Aided-design* Vol. 18, No.3, pp 147—155

[16] Priess, K.(1984). Constructing The Solid Representation From Engineering Projections. *Computer Graphics* Vol. 8, No. 4,pp. 381–389

[17] Aldefeld, B. and Richter, H. (1984). Semianutomatic Three–Dimensional Interpretation of Line Drawing. *Computer Graphics* Vol. 8, No. 4, pp. 371—380

[18] Aldefeld, B. (1983). On Autamatic Recognition of 3D structures from 2D representation. *Computer-Aıded-Design* Vol. 15, No. 2, pp 59—63

[19] Sakurai, H. (1983). Solid Model Input Through Orthographic Views. *Computer Graphics*, Vol. 17, No. 3, pp243—247

[20] Haralick R. M. and Shaprio. (1982). Understanding Engineering Drawing. *Computer Graphics and Image Processing* Vol. 20, pp 244—258

[21] Markowasky, G. and Wesley M. A. (1980). Fleshing Out Wireframes: Reconstruction of Objects,*part i IBM J. Research and Dev.* Vol. 24, No. 5, pp 582—597

[22] Markowasky, G. and Wesley M. A. (1980). Fleshing Out Wireframes: Reconstruction of Objects,*part ii IBM J Research and Dev.* Vol 25, No 6, pp 934—954

[23] Idesawa, M. and Shibata, S.(1975). Automatic Input of Line Drawing and Generation of Solid Figure from Three View Data. *Proc. Int Comp. Symposium*Vol 2, pp 216—225

[24] Shapira R.(1974). A Technique for the Reconstruction of Straight-Edge, Wireframe Object from Two or More Central Projections. *Computer Graphics and Image Processing* Vol. 3, pp 318—326

[25] Zeid, I.(1999) CAD/CAM Theory and Practice. New Delhi, Tata McGraw-Hill.

[26] Mortenson Michael E. (1985). Geometric Modeling. John Willy and sonse.

[27] Bhatt N. D. and Panchal V. M. (1996) Engineering Drawing (Plane and Solid Geometry) Charotar Publication Gujrat.

[28] Lakshminarayana V. and Mathur M. C. (1992) A Text Book of Machine Drawing. Jain Brothers New Delhi.

[29] Khanna, M. L. Solid Geometry. (1967) Merrut, Jai Prakash Nath and Co.

Appendix A
Input Format

Input file should confirm to the following format

Number of vertices in front view (Nv_f)
X and Z coordinates of these Nv_f vertices

Number of vertices in top view (Nv_t)
X and Y coordinates of these Nv_t vertices

Number of vertices in side view (Nv_s)
Y and Z coordinates of these Nv_s vertices

Number of edges in top view (Ne_t)
Number of arcs in top view
Edge numbers of type arc.
Edge Information, vertex index in top vertex list for start and end point of edge
If edge is an arc then center point major axis and minor axis radius and sense.

Number of edges in side view (Ne_s)
Number of arcs in side view
Edge numbers of type arc.
Edge Information, vertex index in top vertex list for start and end point of edge
If edge is an arc then center point major axis and minor axis radius and sense.

Number of edges in front view (Ne_f)
Number of arcs in front view
Edge numbers of type arc.
Edge Information, vertex index in top vertex list for start and end point of edge
If edge is an arc then center point major axis and minor axis radius and sense.

Appendix B
File Oraganization

There are 8 files each contains some specific part of programme. A brief Description of name of file and important functions in it is given here. Name of functions are self explanatory.

1. Display.cpp

This file contains OpenGL related functions. Functions in this file are related to control and define the display variable for OpenGL. These display variables include. Window height, width and position. Some simple drawing functions like line and ellipse drawing are also defined in this file.

2. Define.cpp

This file consist of the **class definitions** used in rest of file. This forms Basis of data structure used. Class defined are:

class point2D	Data structure to store 2D point.
class edge2D	Data structure to store 2D edges.
class vertex3D	Data structure to store 3D vertices.
class edge3D	Data structure to store 3D edges.

3. PreProcessor.cpp

This file define preprocessing functions and input functions.

ReadViews();	Read input
findImplicit();	Generate implicit edges and vertices.
QuickSortSide();	Sort side edges
QuickSortTop();	Sort Top edges

4. EdgeGenerator.cpp

void Case#(int n); # represents [A—N] deals with Cases shown in Figure [3.2, 3.3]
void generateEdges(); Traverse tree shown in figure 3.5 using above functions.

5. RedundatRemoval.cpp

checkOverlapping();	Remove overlapping edges
checkPathological();	Remove Pathological redundant edges

6. FindSurfce.cpp

class cylindricalSurface	Data Structure to store cylindrical surface
class sphericalSurface	Data Structure to store spherical surface
void findSurface();	Find cylindrical and spherical surfaces and store

7. Keyboard.h

Function to listen various keyboard input.

8. ShowResults.cpp

main() in this file calls related functions from above files. Function to display the final results are also defined in this file.

Beside these files there are some more related files.
INPUT_# {# =1,2...30} are input files for the results shown.
README all the information for compiling and running program is given in this.
RCAD is executable file for linux.
MakeRCAD this makefile for software.

Appendix C

Code For Edge Generation Algorithm

```
#include"Preprocessor.cpp"
/*******************************************************
 *                                                     *
 *          FILENAME : EdgeGenerator.cpp               *
 *          Description: This file generate All the possible *
 *              3D edgbes from given 2D Edges          *
 *       functions CaseA(int n) to CaseN(int n) deal   *
 *       with the Cases shown in 3.2 and 3.3 argument 'n' *
 *       is front edge index in front edge list fe[]   *
 *       possible 3d Edges are stored in edgeList[]    *
 *       Function GenerateEdges() generate 3D edges    *
 *                                                     *
 *******************************************************/

void CaseA(int n){
        edge3D temp;
        for(int i=1;i<=Ne_t;i++){
                if(te[i].Case==1){
                        bool c1=fabs(te[i].getExtrema(MIN_ABSSICCA).getAbssicca()
                          -fe[n].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
                        bool c2=fabs(te[i].getExtrema(MAX_ABSSICCA).getAbssicca()
                          -fe[n].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
                        if((c1)AND(c2)){
                                for(int j=1;j<=Nv_s;j++){
                                        if((fabs(te[i].start.getOrdinate()
                                                -sv[j].getAbssicca())<EPSLION)AND
                                          (fabs(fe[n].start.getOrdinate()-
                                                sv[j].getOrdinate())<EPSLION)){
                                                temp.start.setX(te[i].start.getAbssicca());
                                                temp.start.setY(sv[j].getAbssicca());
                                                temp.start.setZ(sv[j].getOrdinate());
                                                temp.finish.setX(te[i].finish.getAbssicca());
                                                temp.finish.setY(sv[j].getAbssicca());
                                                temp.finish.setZ(sv[j].getOrdinate());
                                                temp.frontIndex=n;
                                                temp.sideIndex=j;
                                                temp.topIndex=i;
                                                edgeList[++noEdges]=temp;
                                        }
                                }
                        }
                }
        }
}
```

```
void CaseB(int n){
        edge3D temp;
        for(int i=1;i<=Ne_t;i++){
                if(te[i].Case==3){
                        bool c1=fabs(te[i].getExtrema(MIN_ABSSICCA).getAbssicca()
                          -fe[n].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
                        bool c2=fabs(te[i].getExtrema(MAX_ABSSICCA).getAbssicca()
                          -fe[n].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
                        if((c1)AND(c2)){
                           for(int j=1;j<=Ne_s;j++){
                                if(se[j].Case==5){
                                        bool c1=fabs(te[i].getExtrema(MIN_ORDI
                                        NATE).getOrdinate()-
                                        se[j].getExtrema(MIN_ABSSICCA).
                                        getAbssicca()) <EPSLION;
                                        boolc2=fabs(te[i].getExtrema(MAX_ORDI
                                        NATE).getOrdinate()-
                                        se[j].getExtrema(MAX_ABSSICCA).
                                        getAbssicca())<EPSLION;
                                        bool c3=fabs(fe[n].start.getOrdinate()-
                                        se[j].start.getOrdinate()) <EPSLION;
                                    if((c1)AND(c2)AND(c3)){
                                        temp.start.setX(te[i].start.getAbssicca());
                                        temp.start.setY(te[i].start.getOrdinate());
                                        temp.start.setZ(fe[n].start.getOrdinate());
                                        temp.finish.setX(te[i].finish.getAbssicca());
                                        temp.finish.setY(te[i].finish.getOrdinate());
                                        temp.finish.setZ(fe[n].start.getOrdinate());
                                        temp.frontIndex=n;
                                        temp.sideIndex=j;
                                        temp.topIndex=i;
                                        edgeList[++noEdges]=temp;
                                    }
                                }
                           }
                        }
                }
        }
}

void CaseC(int n){
        edge3D temp;
        for(int i=1;i<=Ne_s;i++){
                if(se[i].Case==2){
                        bool c1=fabs(fe[n].getExtrema(MIN_ORDINATE).getOrdinate()-
                        se[i].getExtrema(MIN_ORDINATE).getOrdinate())<EPSLION;
                        bool c2=fabs(fe[n].getExtrema(MAX_ORDINATE).getOrdinate()-
                        se[i].getExtrema(MAX_ORDINATE).getOrdinate())<EPSLION;
```

```
                        if((c1)AND(c2)){
                                for(int j=1;j<=Nv_t;j++){
                                        c1=fabs(fe[n].start.getAbssicca()-
                                        tv[j].getAbssicca())<EPSLION;
                                        c2=fabs(se[i].start.getAbssicca()-
                                        tv[j].getOrdinate())<EPSLION;
                                        if((c1)AND(c2)){
                                                temp.start.setX(tv[j].getAbssicca());
                                                temp.start.setY(tv[j].getOrdinate());
                                                temp.start.setZ(fe[n].start.getOrdinate());
                                                temp.finish.setX(tv[j].getAbssicca());
                                                temp.finish.setY(tv[j].getOrdinate());
                                                temp.finish.setZ(fe[n].finish.getOrdinate());
                                                temp.frontIndex=n;
                                                temp.sideIndex=i;
                                                temp.topIndex=j;
                                                edgeList[++noEdges]=temp;
                                        }
                                }
                        }
                }
        }
}

void CaseD(int n){
        edge3D temp;
        for(int i=1;i<=Ne_t;i++){
                if(te[i].Case==5){
                    if(fabs(fe[n].start.getAbssicca()
                       -te[i].start.getAbssicca())<EPSLION){
                        for(int j=1;j<=Ne_s;j++){
                            if(se[j].Case==3){
                                bool c1=fabs(se[j].getExtrema(MIN_ABSSICCA).
                                        getAbssicca()-
                                        te[i].getExtrema(MIN_ORDINATE).
                                        getOrdinate())<EPSLION;
                                bool c2=fabs(se[j].getExtrema(MAX_ABSSICCA).
                                        getAbssicca()-
                                    te[i].getExtrema(MAX_ORDINATE).
                                        getOrdinate())<EPSLION;
                                bool c3=fabs(se[j].getExtrema(MIN_ORDINATE).
                                        getOrdinate() -
                                        fe[n].getExtrema(MIN_ORDINATE).
                                        getOrdinate())<EPSLION;
                                bool c4=fabs(se[j].getExtrema(MAX_ORDINATE)
                                        .getOrdinate() -
                                        fe[n].getExtrema(MAX_ORDINATE)
                                        .getOrdinate())<EPSLION;
                                if((c1)AND(c2)AND(c3)AND(c4)){
```

```
                                        temp.start.setX(fe[n].start.getAbssicca());
                                        temp.start.setY(se[j].start.getAbssicca());
                                        temp.start.setZ(se[j].start.getOrdinate());
                                        temp.finish.setX(fe[n].start.getAbssicca());
                                        temp.finish.setY(se[j].finish.getAbssicca());
                                        temp.finish.setZ(se[j].finish.getOrdinate());
                                        temp.frontIndex=n;
                                        temp.sideIndex=j;
                                        temp.topIndex=i;
                                        edgeList[++noEdges]=temp;
                                    }
                                }
                            }
                        }
                    }
                }
        }
}

void CaseE(int n){
        edge3D temp;
        for(int i=1;i<=Ne_t;i++){
                if(te[i].Case==1){
                        bool c1=fabs(fe[n].getExtrema(MIN_ABSSICCA).getAbssicca()-
                        te[i].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
                        bool c2=fabs(fe[n].getExtrema(MAX_ABSSICCA).getAbssicca()-
                        te[i].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
                        if((c1)AND(c2)){
                                for(int j=1;j<=Ne_s;j++){
                                  if(se[j].Case==2){
                                  c1=fabs(fe[n].getExtrema(MIN_ORDINATE).
                                        getOrdinate()-
                                        se[j].getExtrema(MIN_ORDINATE).
                                        getOrdinate())<EPSLION;
                                  c2=fabs(fe[n].getExtrema(MAX_ORDINATE).
                                        getOrdinate()-
                                        se[j].getExtrema(MAX_ORDINATE).
                                        getOrdinate())<EPSLION;
                                  bool c3=fabs(se[j].getExtrema(MIN_ABSSICCA).
                                        getAbssicca()-
                                        te[i].getExtrema(MIN_ORDINATE).
                                        getOrdinate())<EPSLION;
                                  if((c1)AND(c2)AND(c3)){
                                                temp.start.setX(fe[n].start.getAbssicca());
                                                temp.start.setZ(fe[n].start.getOrdinate());
                                                temp.start.setY(te[i].start.getOrdinate());
                                                temp.finish.setX(fe[n].finish.getAbssicca());
                                                temp.finish.setZ(fe[n].finish.getOrdinate());
                                                temp.finish.setY(te[i].start.getOrdinate());
                                                temp.frontIndex=n;
```

```
                                        temp.sideIndex=j;
                                        temp.topIndex=i;
                                        edgeList[++noEdges]=temp;
                                        temp.write();
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    void CaseF(int n){
        edge3D temp;
        for(int i=1;i<=Ne_t;i++){
            if(te[i].Case==3){
                bool c1=fabs(fe[n].getExtrema(MIN_ABSSICCA).getAbssicca()-
                    te[i].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
                bool c2=fabs(fe[n].getExtrema(MAX_ABSSICCA).getAbssicca()-
                    te[i].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
                if((c1)AND(c2)){
                    for(int j=1;j<=Ne_s;j++){
                      if(se[j].Case==3){
                    c1=fabs(fe[n].getExtrema(MIN_ORDINATE).
                            getOrdinate()-
                            se[j].getExtrema(MIN_ORDINATE).
                            getOrdinate())<EPSLION;
                    c2=fabs(fe[n].getExtrema(MAX_ORDINATE).
                            getOrdinate()-
                            se[j].getExtrema(MAX_ORDINATE).
                            getOrdinate())<EPSLION;
                     bool c3=fabs(se[j].getExtrema(MIN_ABSSICCA).
                            getAbssicca()-
                            te[i].getExtrema(MIN_ORDINATE).
                            getOrdinate())<EPSLION;
                     bool c4=fabs(se[j].getExtrema(MAX_ABSSICCA).
                            getAbssicca()-
                            te[i].getExtrema(MAX_ORDINATE).
                            getOrdinate())<EPSLION;
                     if((c1)AND(c2)AND(c3)AND(c4)){
                            temp.start.setX(fe[n].getExtrema
                                (MIN_ABSSICCA).getAbssicca());
                            temp.start.setY(te[i].getExtrema
                                (MIN_ABSSICCA).getOrdinate());
                            temp.start.setZ(fe[n].getExtrema
                                (MIN_ABSSICCA).getOrdinate());
                            temp.finish.setX(fe[n].getExtrema
                                (MAX_ABSSICCA).getAbssicca());
```

```
                                        temp.finish.setY(te[i].getExtrema
                                            (MAX_ABSSICCA).getOrdinate());
                                        temp.finish.setZ(fe[n].getExtrema
                                            (MAX_ABSSICCA).getOrdinate());
                                        temp.frontIndex=n;
                                        temp.sideIndex=j;
                                        temp.topIndex=i;
                                        edgeList[++noEdges]=temp;
                                    }
                                }
                            }
                        }
                    }
                }
            }

void CaseG(int n){
        edge3D temp;
        for(int i=1;i<=Ne_t;i++){
                if(te[i].Case==5){
                  if(fabs(fv[n].getAbssicca()-te[i].start.getAbssicca())<EPSLION){
                        for(int j=1;j<=Ne_s;j++){
                                if(se[j].Case==5){
                                        bool c1=fabs(fv[n].getOrdinate()-
                                                se[j].start.getOrdinate())<EPSLION;
                                        bool c2=fabs(se[j].getExtrema(MIN_ABSSICCA).
                                                getAbssicca()-
                                            te[i].getExtrema(MIN_ORDINATE).
                                                getOrdinate())<EPSLION;
                                        bool c3=fabs(se[j].getExtrema(MAX_ABSSICCA).
                                                getAbssicca()-
                                            te[i].getExtrema(MAX_ORDINATE).
                                                getOrdinate())<EPSLION;
                                        if((c1)AND(c2)AND(c3)){
                                                temp.start.setX(fv[n].getAbssicca());
                                                temp.start.setZ(fv[n].getOrdinate());
                                                temp.start.setY(te[i].start.getOrdinate());
                                                temp.finish.setX(fv[n].getAbssicca());
                                                temp.finish.setZ(fv[n].getOrdinate());
                                                temp.finish.setY(te[i].finish.getOrdinate());
                                            temp.frontIndex=n;
                                            temp.sideIndex=j;
                                            temp.topIndex=i;
                                            edgeList[++noEdges]=temp;
                                        }
                                }
                        }
                  }
                }
        }
```

```
        }
}

void CaseH(int n){
        edge3D temp;
        static int v;
        for(int i=1;i<=Ne_t;i++){
        if(te[i].Case==1){
                bool c1=fabs(fe[n].getExtrema(MIN_ABSSICCA).getAbssicca()-
                     te[i].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
                bool c2=fabs(fe[n].getExtrema(MAX_ABSSICCA).getAbssicca()-
                        te[i].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
                if((c1)AND(c2)){
                  for(int j=1;j<=Ne_s;j++){
                        if(se[j].Case==2){
                                c1=fabs(fe[n].getExtrema(MIN_ORDINATE).
                                        getOrdinate()-
                                        se[j].getExtrema(MIN_ORDINATE).
                                        getOrdinate())<EPSLION;
                                c2=fabs(fe[n].getExtrema(MAX_ORDINATE).
                                        getOrdinate()-
                                        se[j].getExtrema(MAX_ORDINATE).
                                        getOrdinate())<EPSLION;
                                bool c3=fabs(se[j].start.getAbssicca()-
                                         te[i].start.getOrdinate())<EPSLION;
                                if((c1)AND(c2)AND(c3)){
                                        temp.type=TYPE_ELLIPS;
                                        temp.start.setX(fe[n].start.getAbssicca());
                                        temp.start.setY(te[i].start.getOrdinate());
                                        temp.start.setZ(fe[n].start.getOrdinate());
                                        temp.finish.setX(fe[n].finish.getAbssicca());
                                        temp.finish.setY(te[i].finish.getOrdinate());
                                        temp.finish.setZ(fe[n].finish.getOrdinate());
                                        temp.center.setX(fe[n].center.getAbssicca());
                                        temp.center.setY(te[i].start.getOrdinate());
                                        temp.center.setZ(fe[n].center.getOrdinate());
                                        temp.a=fe[n].a;
                                        temp.b=fe[n].b;
                                        temp.startAngle=fe[n].startAngle;
                                        temp.finishAngle=fe[n].finishAngle;
                                        temp.sense=fe[n].sense;
                                        temp.dc[0]=90;
                                        temp.dc[1]=0;
                                        temp.dc[2]=0;
                                        temp.frontIndex=n;
                                        temp.topIndex=i;
                                        temp.sideIndex=j;
                                        edgeList[++noEdges]=temp;
                                }
```

```
                    }
                }
            }
        }
    }
}

void CaseI(int n){
    edge3D temp;
    for(int i=1;i<=Ne_t;i++){
     if(te[i].Case==3){
        bool c1=fabs(fe[n].getExtrema(MIN_ABSSICCA).getAbssicca()-
            te[i].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
        bool c2=fabs(fe[n].getExtrema(MAX_ABSSICCA).getAbssicca()-
            te[i].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
        if((c1)AND(c2)){
          for(int j=1;j<=Ne_s;j++){
            if(se[j].Case==6){
              c1=fabs(fe[n].getExtrema(MIN_ORDINATE).
                    getOrdinate()-
                    se[j].getExtrema(MIN_ORDINATE).
                    getOrdinate())<EPSLION;
              c2=fabs(fe[n].getExtrema(MAX_ORDINATE).
                    getOrdinate()-
                    se[j].getExtrema(MAX_ORDINATE).
                    getOrdinate())<EPSLION;
              bool c3=fabs(te[i].getExtrema(MAX_ORDINATE).
                    getOrdinate()-
                    se[j].getExtrema(MAX_ABSSICCA).
                    getAbssicca())<EPSLION;
              bool c4=fabs(te[i].getExtrema(MIN_ORDINATE).
                    getOrdinate()-
                    se[j].getExtrema(MIN_ABSSICCA).
                    getAbssicca())<EPSLION;
              if((c1)AND(c2)AND(c3)AND(c4)){
                    temp.type=TYPE_ELLIPS;
                    temp.start.setX(fe[n].start.getAbssicca());
                    temp.start.setY(se[j].start.getAbssicca());
                    temp.start.setZ(fe[n].start.getOrdinate());
                    temp.finish.setX(fe[n].finish.getAbssicca());
                    temp.finish.setY(se[j].finish.getAbssicca());
                    temp.finish.setZ(fe[n].finish.getOrdinate());
                    temp.center.setX(fe[n].center.getAbssicca());
                    temp.center.setY(se[j].center.getAbssicca());
                    temp.center.setZ(fe[n].center.getOrdinate());
                    temp.startAngle=fe[n].startAngle;
                    temp.finishAngle=fe[n].finishAngle;
                    float tmpang1=angle(te[i].getExtrema
                          (MIN_ORDINATE).getAbssicca(),
```

```
                                te[i].getExtrema(MIN_ORDINATE).getOrdinate(),
                                        te[i].getExtrema(MAX_ORDINATE).
                                            getAbssicca(),
                                        te[i].getExtrema(MAX_ORDINATE).
                                                    getOrdinate());
                                float tmpang2=tmpang1;
                                while(tmpang2>90)
                                tmpang2=(tmpang2>=90? tmpang2-90:tmpang2);
                                temp.b=fe[n].b;
                                temp.a=fe[n].a/cos(tmpang2*M_PI/180.0);
                                temp.sense=fe[n].sense;
                                temp.dc[1]=tmpang1;
                                temp.dc[0]=90;
                                temp.dc[2]=0;
                                temp.frontIndex=n;
                                temp.topIndex=i;
                                temp.sideIndex=j;
                                edgeList[++noEdges]=temp;
                            }
                        }
                    }
                }
            }
        }
}

void CaseJ(int n){
        edge3D temp;
        for(int i=1;i<=Ne_t;i++){
         if(te[i].Case==6){
                bool c1=fabs(fe[n].getExtrema(MIN_ABSSICCA).getAbssicca()-
                     te[i].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
                bool c2=fabs(fe[n].getExtrema(MAX_ABSSICCA).getAbssicca()-
                        te[i].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
                if((c1)AND(c2)){
                   for(int j=1;j<=Ne_s;j++){
                        if(se[j].Case==3){
                                c1=fabs(fe[n].getExtrema(MIN_ORDINATE).
                                        getOrdinate()-
                                         se[j].getExtrema(MIN_ORDINATE).
                                        getOrdinate())<EPSLION;
                                c2=fabs(fe[n].getExtrema(MAX_ORDINATE).
                                        getOrdinate()-
                                        se[j].getExtrema(MAX_ORDINATE).
                                        getOrdinate())<EPSLION;
                                bool c3=fabs(te[i].getExtrema(MAX_ORDINATE).
                                        getOrdinate()-
                                         se[j].getExtrema(MAX_ABSSICCA).
                                        getAbssicca())<EPSLION;
```

```
                    bool c4=fabs(te[i].getExtrema
                            (MIN_ORDINATE).getOrdinate()-
                             se[j].getExtrema
                            (MIN_ABSSICCA).getAbssicca())<EPSLION;
                    if((c1)AND(c2)AND(c3)AND(c4)){
                            temp.type=TYPE_ELLIPS;
                            temp.start.setX(fe[n].start.getAbssicca());
                            temp.start.setY(te[i].start.getOrdinate());
                            temp.start.setZ(fe[n].start.getOrdinate());
                            temp.finish.setX(fe[n].finish.getAbssicca());
                            temp.finish.setY(te[i].finish.getOrdinate());
                            temp.finish.setZ(fe[n].finish.getOrdinate());
                            temp.center.setX(fe[n].center.getAbssicca());
                            temp.center.setY(te[i].center.getOrdinate());
                            temp.center.setZ(fe[n].center.getOrdinate());
                            temp.startAngle=te[i].startAngle;
                            temp.finishAngle=te[i].finishAngle;
                            float tmpang1=angle(se[j].getExtrema
                                    (MIN_ORDINATE).getAbssicca(),
                                            se[j].getExtrema
                                            (MIN_ORDINATE).getOrdinate(),
                                            se[j].getExtrema
                                            (MAX_ORDINATE).getAbssicca(),
                                            se[j].getExtrema
                                            (MAX_ORDINATE).getOrdinate());
                            float tmpang2=tmpang1;
                            while(tmpang2>90)
                            tmpang2=(tmpang2>=90? tmpang2-90:tmpang2);
                            temp.a=fe[n].a;
                            temp.b=fe[n].b/cos(tmpang2*M_PI/180);
                            temp.sense=fe[n].sense;
                            temp.dc[0]=tmpang1;
                            temp.dc[1]=0;
                            temp.dc[2]=0;
                            temp.frontIndex=n;
                            temp.topIndex=i;
                            temp.sideIndex=j;
                            edgeList[++noEdges]=temp;
                    }
                }
            }
        }
    }
}

void CaseK(int n){
    edge3D temp;
    for(int i=1;i<=Ne_t;i++){
```

```
if(te[i].Case==6){
        bool c1=fabs(fe[n].getExtrema(MIN_ABSSICCA).getAbssicca()-
             te[i].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
        bool c2=fabs(fe[n].getExtrema(MAX_ABSSICCA).getAbssicca()-
              te[i].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
        if((c1)AND(c2)){
           for(int j=1;j<=Ne_s;j++){
                if(se[j].Case==6){
                      c1=fabs(fe[n].getExtrema(MIN_ORDINATE).
                            getOrdinate()-
                            se[j].getExtrema(MIN_ORDINATE).
                            getOrdinate())<EPSLION;
                      c2=fabs(fe[n].getExtrema(MAX_ORDINATE).
                            getOrdinate()-
                             se[j].getExtrema(MAX_ORDINATE).
                            getOrdinate())<EPSLION;
                      bool c3=fabs(te[i].getExtrema(MAX_ORDINATE).
                            getOrdinate()-
                             se[j].getExtrema(MAX_ABSSICCA).
                            getAbssicca())<EPSLION;
                      bool c4=fabs(te[i].getExtrema(MIN_ORDINATE).
                            getOrdinate()-
                             se[j].getExtrema(MIN_ABSSICCA).
                            getAbssicca())<EPSLION;
                    if((c1)AND(c2)AND(c3)AND(c4)){
                            temp.type=TYPE_ELLIPS;
                            temp.start.setX(te[i].start.getAbssicca());
                            temp.start.setY(te[i].start.getOrdinate());
                            temp.start.setZ(se[j].start.getOrdinate());
                            temp.finish.setX(te[i].finish.getAbssicca());
                            temp.finish.setY(te[i].finish.getOrdinate());
                            temp.finish.setZ(se[j].finish.getOrdinate());
                            temp.center.setX(te[i].center.getAbssicca());
                            temp.center.setY(te[i].center.getOrdinate());
                            temp.center.setZ(se[j].center.getOrdinate());
                            temp.startAngle=te[i].startAngle;
                            temp.finishAngle=te[i].finishAngle;
                            float tmpang1=angle(fe[n].getExtrema
                                  (MIN_ORDINATE).getAbssicca(),
                                        fe[n].getExtrema
                                        (MIN_ORDINATE).getOrdinate(),
                                        fe[n].getExtrema
                                        (MAX_ORDINATE).getAbssicca(),
                                        fe[n].getExtrema
                                        (MAX_ORDINATE).getOrdinate());
                           float tmpang2=tmpang1;
                           while(tmpang2>90)
                           tmpang2=(tmpang2>=90? tmpang2-90:tmpang2);
                           temp.b=te[i].b;
```

```
                                temp.a=te[i].a/cos(tmpang2*M_PI/180);
                                temp.sense=!se[j].sense;
                                temp.dc[1]=-tmpang1;
                                temp.dc[0]=0;
                                temp.dc[2]=0;
                                temp.frontIndex=n;
                                temp.topIndex=i;
                                temp.sideIndex=j;
                                edgeList[++noEdges]=temp;
                            }
                        }
                    }
                }
            }
        }
}

void Case1(int n){
        edge3D temp;
        for(int i=1;i<=Ne_t;i++){
        if(te[i].Case==6){
                bool c1=fabs(fe[n].getExtrema(MIN_ABSSICCA).getAbssicca()-
                    te[i].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
                bool c2=fabs(fe[n].getExtrema(MAX_ABSSICCA).getAbssicca()-
                    te[i].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
                if((c1)AND(c2)){
                  for(int j=1;j<=Ne_s;j++){
                        if(se[j].Case==5){
                                c1=fabs(te[i].getExtrema(MIN_ORDINATE).getOrdinate()-
                                    se[j].getExtrema(MIN_ABSSICCA).
                                    getAbssicca())<EPSLION;
                                c2=fabs(te[i].getExtrema(MAX_ORDINATE).
                                    getOrdinate()-
                                    se[j].getExtrema(MAX_ABSSICCA).
                                    getAbssicca())<EPSLION;
                                bool c3=fabs(se[j].start.getOrdinate()-
                                    f   e[n].start.getOrdinate())<EPSLION;
                                if((c1)AND(c2)AND(c3)){
                                    temp.type=TYPE_ELLIPS;
                                    temp.start.setX(te[i].start.getAbssicca());
                                    temp.start.setY(te[i].start.getOrdinate());
                                    temp.start.setZ(fe[n].start.getOrdinate());
                                    temp.finish.setX(te[i].finish.getAbssicca());
                                    temp.finish.setY(te[i].finish.getOrdinate());
                                    temp.finish.setZ(fe[n].finish.getOrdinate());
                                    temp.center.setX(te[i].center.getAbssicca());
                                    temp.center.setY(te[i].center.getOrdinate());
                                    temp.center.setZ(fe[n].start.getOrdinate());
```

```
                                temp.a=te[i].a;
                                temp.b=te[i].b;
                                temp.startAngle=te[i].startAngle;
                                temp.finishAngle=te[i].finishAngle;
                                temp.sense=te[i].sense;
                                temp.dc[0]=0;
                                temp.dc[1]=0;
                                temp.dc[2]=0;
                                temp.frontIndex=n;
                                temp.topIndex=i;
                                temp.sideIndex=j;
                                edgeList[++noEdges]=temp;
                            }
                        }
                    }
                }
            }
        }
}

void CaseM(int n){
        edge3D temp;
        for(int i=1;i<=Ne_t;i++){
        if(te[i].Case==5){
                bool c1=fabs(fe[n].getExtrema(MIN_ABSSICCA).getAbssicca()-
                        te[i].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
                bool c2=fabs(fe[n].getExtrema(MAX_ABSSICCA).getAbssicca()-
                        te[i].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
                if((c1)AND(c2)){
                    for(int j=1;j<=Ne_s;j++){
                        if(se[j].Case==6){
                            c1=fabs(te[i].getExtrema(MIN_ORDINATE).
                                    getOrdinate()-
                                    se[j].getExtrema(MIN_ABSSICCA).
                                    getAbssicca())<EPSLION;
                            c2=fabs(te[i].getExtrema
                                    (MAX_ORDINATE).getOrdinate()-
                                    se[j].getExtrema(MAX_ABSSICCA).
                                    getAbssicca())<EPSLION;
                            bool c3=fabs(te[i].getExtrema(MAX_ORDINATE).
                                    getOrdinate()-
                                    se[j].getExtrema(MAX_ABSSICCA).
                                    getAbssicca())<EPSLION;
                            bool c4=fabs(te[i].getExtrema(MIN_ORDINATE).
                                    getOrdinate()-
                                    se[j].getExtrema(MIN_ABSSICCA).
                                    getAbssicca())<EPSLION;
                            if((c1)AND(c2)AND(c3)AND(c4)){
                                    temp.type=TYPE_ELLIPS;
```

```
                                        temp.start.setX(te[i].start.getAbssicca());
                                        temp.start.setY(se[j].start.getAbssicca());
                                        temp.start.setZ(se[j].start.getOrdinate());
                                        temp.finish.setX(te[i].finish.getAbssicca());
                                        temp.finish.setY(se[j].finish.getAbssicca());
                                        temp.finish.setZ(se[j].finish.getOrdinate());
                                        temp.center.setX(te[i].start.getAbssicca());
                                        temp.center.setY(se[j].center.getAbssicca());
                                        temp.center.setZ(se[j].center.getOrdinate());
                                        temp.a=se[j].a;
                                        temp.b=se[j].b;
                                        temp.startAngle=se[j].startAngle;
                                        temp.finishAngle=se[j].finishAngle;
                                        temp.sense=se[j].sense;
                                        temp.dc[0]=90;
                                        temp.dc[1]=90;
                                        temp.dc[2]=0;
                                        temp.frontIndex=n;
                                        temp.topIndex=i;
                                        temp.sideIndex=j;
                                        edgeList[++noEdges]=temp;
                                }
                        }
                }
        }
}

void CaseN(int n){
        edge3D temp;
        for(int i=1;i<=Ne_t;i++){
         if(te[i].Case==6){
                bool c1=fabs(fe[n].getExtrema(MIN_ABSSICCA).getAbssicca()-
                        te[i].getExtrema(MIN_ABSSICCA).getAbssicca())<EPSLION;
                bool c2=fabs(fe[n].getExtrema(MAX_ABSSICCA).getAbssicca()-
                        te[i].getExtrema(MAX_ABSSICCA).getAbssicca())<EPSLION;
                bool c3=fabs(te[i].center.getAbssicca()-
                        fe[n].center.getAbssicca())<EPSLION;
                if((c1)AND(c2)AND(c3)){
                   for(int j=1;j<=Ne_s;j++){
                        if(se[j].Case==6){
                                c1=fabs(fe[n].getExtrema(MIN_ORDINATE).
                                        getOrdinate()-
                                  se[j].getExtrema(MIN_ORDINATE).
                                        getOrdinate())<EPSLION;
                                c2=fabs(fe[n].getExtrema(MAX_ORDINATE).
                                        getOrdinate()-
                                        se[j].getExtrema(MAX_ORDINATE).
```

```
        getOrdinate())<EPSLION;
    bool c3=fabs(te[i].getExtrema(MAX_ORDINATE).
        getOrdinate()-
        se[j].getExtrema(MAX_ABSSICCA).
        getAbssicca())<EPSLION;
    bool c4=fabs(te[i].getExtrema(MIN_ORDINATE).
        getOrdinate()-
        se[j].getExtrema(MIN_ABSSICCA).
        getAbssicca())<EPSLION;
    bool c5=fabs(te[i].center.getOrdinate()-
                se[j].center.getAbssicca())<EPSLION;
    bool c6=fabs(fe[n].center.getOrdinate()-
                se[j].center.getOrdinate())<EPSLION;;
if((c1)AND(c2)AND(c3)AND(c4)AND(c5)AND(c6)){
        temp.type=TYPE_ELLIPS;
        temp.start.setX(te[i].getExtrema
                (MIN_ABSSICCA).getAbssicca());
        temp.start.setY(te[i].getExtrema
                (MIN_ABSSICCA).getOrdinate());
        temp.start.setZ(fe[n].getExtrema
                (MIN_ABSSICCA).getOrdinate());
        temp.finish.setX(te[i].getExtrema
                (MAX_ABSSICCA).getAbssicca());
        temp.finish.setY(te[i].getExtrema
                (MAX_ABSSICCA).getOrdinate());
        temp.finish.setZ(fe[n].getExtrema
                (MAX_ABSSICCA).getOrdinate());
        temp.center.setX(te[i].center.getAbssicca());
        temp.center.setY(te[i].center.getOrdinate());
        temp.center.setZ(fe[n].center.getOrdinate());
        temp.startAngle=te[i].startAngle;
        temp.finishAngle=te[i].finishAngle;
        temp.sense=te[i].sense;
        float tempang1=angle(0,0,
                temp.center.getY(),temp.center.getZ());
        float tempang2=angle(0,0,
                temp.center.getX(),temp.center.getZ());
        float tempang3=tempang1;
        while(tempang3>90)
        tempang3=(tempang3>=90?
                tempang3-90:tempang3);
        float tempang4=tempang2;
        while(tempang4>90)
        tempang4=(tempang4>=90?
                tempang4-90:tempang4);
        temp.a=te[i].a;
        temp.b=fe[n].b;
        temp.dc[1]=-tempang1;
        temp.dc[0]=tempang2;
```

```
                                        temp.dc[2]=0;
                                        temp.frontIndex=n;
                                        temp.topIndex=i;
                                        temp.sideIndex=j;
                                        edgeList[++noEdges]=temp;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    void GenerateEdges(){
        edgeList=new edge3D[Ne_t*Ne_f];
        for(int i=1;i<=Ne_f;i++){
            switch(fe[i].Case){
                case 1:
                    CaseA(i);
                    CaseB(i);
                    CaseL(i);
                    break;
                case 2:
                    CaseC(i);
                    CaseD(i);
                    CaseM(i);
                    break;
                case 3:
                    CaseE(i);
                    CaseF(i);
                    CaseK(i);
                    break;
                case 6:
                    CaseH(i);
                    CaseI(i);
                    CaseJ(i);
                    CaseN(i);
                    break;
                default:
                    cout<<"\n\n:ERROR OOCURED IN
                        `void GenerateEdges()':\n";
                    cout<<"PLEASE CHECK YOUR INPUT
                        @QUITING NOW @\n\n"<<endl;
                    exit(-1);
            }
        }
        for(int i=1;i<=Nv_f;i++){
            CaseG(i);
        }
    }
```